

Apple

# DOS Programmer's Tool Kit

For the Apple II Family



**WORK-  
BENCH**

# Table of Contents

<b>List of Figures and Tables</b>	<b>ix</b>
-----------------------------------	-----------

<b>Preface</b>	<b>xi</b>
----------------	-----------

xii	What You Need
xiii	Guides to Programming the Apple II
xiii	Reference Manuals on the 6502 Microprocessor
xiv	Other Accessories

## 1

<b>Introduction to Assembly-Language Programming</b>	<b>1</b>
--	----------

3	The Programming Process
5	Creating an Assembly-Language Source File
5	Assembling Your Program
5	Testing and Verifying Your Program
6	Running Assembly-Language Programs Directly From BASIC
6	Running Assembly-Language Programs From a BASIC Program
6	Advanced Techniques—The Apple IIe Identification Routines
6	A Note on the Tutorials

## 2

<b>The Editor</b>	<b>8</b>
-------------------	----------

11	A Tutorial: Editing Text Files
12	Getting Started
14	The Editor Command Level

15	Entering Text With the Editor
16	Displaying Text
17	Line-Editing Text
18	Storing and Retrieving Files
19	Writing a Program With the Editor
21	Leaving the Editor
21	Using the Editor
22	The Editor Command Level
22	Getting Help
23	Typing Uppercase and Lowercase Characters
24	Executing Direct DOS Commands
24	Saving and Retrieving Files on a Disk
29	Manipulating Lines in the Text Buffer
32	Viewing Your Text in the Text Buffer
34	Changing Text Within a Line
41	Editing Two Files at Once
42	Altering the Display
44	Leaving the Editor and Returning to BASIC

### 3

## *The 6502 Assembler*

47

51	Introduction to the Assembler
52	A Tutorial: Using the Assembler
52	Getting Started
53	Assembling Your Program
55	Using the Assembler
55	Calling the Assembler
57	Recovering From Errors
58	Stopping Your Assembly
59	The ASMIDSTAMP File
59	Generating Assembly Listings
65	Assembly-Language Source Files
65	The Syntax of Assembly Statements
73	Giving Directions to the Assembler
74	Controlling the Overall Assembly
78	Assigning Information
81	Generating Data in Your Object Code
85	Controlling Conditional Assembly
88	Source File Directives
91	Controlling Your Assembly Listings
96	Using Macros in Your Assembly-Language Programs
97	Calling Macros in Your Program Source File
97	The Macro Definition File

## **4**

### ***The Bugbyter Debugger***

**102**

- 106 Introduction to Bugbyter
- 107 Restrictions on Using Bugbyter
- 108 A Tutorial: Using Bugbyter
- 109 Getting Started
- 117 Loading Your Program
- 119 Single-Stepping Through Your Program
- 122 Using the Memory Subdisplay
- 125 Tracing Your Program
- 126 Changing Your Program in Memory
- 127 Viewing a Page of Memory
- 129 Using Bugbyter
- 130 Relocating the Bugbyter Program
- 130 Using Bugbyter in a Language Card
- 131 Entering the Monitor
- 131 Restarting Bugbyter
- 132 Memory and the Bugbyter Displays
- 132 Using the Memory Subdisplay
- 133 Viewing the Memory Page Display
- 135 Altering the Contents of Memory
- 136 Altering the Contents of Registers
- 137 Altering Bugbyter's Master Display Layout
- 139 Controlling the Execution of Your Program
- 139 Using the Single-Step and Trace Modes
- 148 Using Execution Mode
- 150 Debugging Real-Time Code
- 152 Debugging Programs that Use the Keyboard and Display
- 156 Executing Undefined Opcodes

## **5**

### ***The Relocating Loader***

**157**

- 160 Introduction to the Relocating Loader
- 160 Restrictions on Using the Relocating Loader
- 161 Using the Relocating Loader
- 161 Calling RBOOT
- 162 Using RLOAD

## **6**

### ***The Apple IIe System Identification Routines***

**165**

- 168 Introduction to the Identification Routine
- 168 Using the Routines in Your Assembly-Language Programs
- 169 Overview of the Identification Procedure



## **Appendixes**

**172**

### **An Editor Quick Guide**

**175**

#### **A**

- 175 Editor Commands: A Functional Summary
- 175 Executing Direct DOS Commands
- 175 Storing and Retrieving Files From a Disk
- 176 Manipulating Lines in the Text Buffer
- 176 Viewing Your Text in the Text Buffer
- 177 Changing Text Within a Line
- 177 Editing Two Files at Once
- 177 Altering the Display
- 178 Leaving the Editor
- 178 Calling the Assembler
- 179 Editor Commands: An Alphabetic Summary
- 182 Edit Mode Keystroke Summary

### **6502 Assembly Language Summary**

**183**

#### **B**

- 183 Mnemonic Summary
- 185 Addressing Mode Summary
- 185 Assembler Directive Summary

### **A Bugbyter Quick Guide**

**187**

#### **C**

- 187 Bugbyter Command Level
- 187 General Commands
- 187 Command-Line Editing
- 188 Memory Reference
- 189 Register Reference
- 189 Trace/Single-Step Mode
- 190 Disassembly Options for Trace/Single-Step
- 190 Breakpoints
- 191 Debugging in Execution Mode
- 191 User Soft Switches

### **Error Messages**

**193**

#### **D**

- 193 Editor Error Messages
- 193 Editor's DOS-Error Messages
- 195 Editor Command-Error Messages
- 197 Assembler Error Messages
- 197 Assembler's DOS-Error Messages
- 199 Assembler Syntax-Error Messages

<b>E</b>	<b>Object File and Symbol Table Formats</b>	<b>207</b>
	207 Object File Format	
	209 Symbol Table Formats	
<b>F</b>	<b>Editing BASIC Programs</b>	<b>211</b>
	211 Using the Editor to Edit BASIC Programs	
<b>G</b>	<b>The SWEET16 Interpreter</b>	<b>213</b>
	213 A 16-Bit Pseudomachine	
	214 SWEET16 Interpreter Operation	
	215 SWEET16 Instruction Summary	
	215 Register Instructions	
	215 Control Instructions	
<b>H</b>	<b>System Memory Usage</b>	<b>217</b>
	217 The Editor/Assembler	
	218 The Bugbyter Debugger	
<b>I</b>	<b>Editor/Assembler File Components</b>	<b>219</b>
	219 The EDASM Files	
	220 Making a HELLO Program	
	<b>Index</b>	<b>221</b>
	<b>Bugbyter Quick Guide</b>	
	<b>Editor and Assembler Quick Reference Guide</b>	

## **List of Figures and Tables**

### **Chapter 1**

- 4** Figure 1-1 The Process of Assembly-Language Programming

### **Chapter 2**

- 14** Figure 2-1 Editor Features Accessible From the Editor Command Level  
**40** Table 2-1 Edit Mode Keystroke Summary

### **Chapter 3**

- 62** Figure 3-1 A Typical Assembly Listing  
**65** Figure 3-2 A Sample Symbol Table Listing

### **Chapter 4**

- 109** Figure 4-1 Bugbyter's Master Display  
**110** Figure 4-2 The Register Subdisplay  
**111** Figure 4-3 The Stack Subdisplay  
**112** Figure 4-4 The Code Disassembly Subdisplay  
**113** Figure 4-5 The Memory Cell Subdisplay  
**113** Figure 4-6 The Breakpoint Subdisplay  
**114** Figure 4-7 Bugbyter Command Line Allows You Access to All Features  
**117** Table 4-1 Bugbyter Command-Line Editing Functions  
**140** Table 4-2 Trace and Single-Step Keystroke Commands  
**143** Figure 4-8 The Breakpoint Subdisplay  
**146** Table 4-3 Code Disassembly Display Options  
**151** Table 4-4 Bugbyter Real-Time Soft Switches

### **Chapter 5**

- 161** Figure 5-1 Using RBOOT and RLOAD Routines in Your BASIC Program

## Chapter 6

- 168 Table 6-1 Identification Procedure Return Values

## Appendixes

- 207 Table E-1 Relocatable File Format  
217 Figure H-1 Memory Map of the 64K Editor/Assembler  
218 Figure H-2 Memory Map Showing Locations of Bugbyter and Your Program

## **Preface**

The DOS Programmer's Tool Kit Volume II disk contains everything you need to program your Apple II system in its native language. These 6502 assembly-language programming tools include

- an Editor—to help you create assembly-language source programs
- an Assembler—to translate your assembly-language source programs into executable 6502 object programs
- the Bugbyter Debugger—a powerful debugging tool to assist you in testing and debugging your programs
- a Relocating Loader—to allow you to load and execute your assembly-language programs during the execution of a BASIC program
- two Identification Routines—to allow your BASIC and assembly-language programs to identify whether they are executing on an Apple II, Apple II Plus, or Apple IIe system.

This manual explains how you can use these tools to create and execute assembly-language programs on your Apple II, Apple II Plus, or Apple IIe computer. In separate chapters, each of these programming tools—the Editor, the Assembler, and the Bugbyter—is described in detail. Each chapter consists of

- an introduction to the programming tool
- a tutorial demonstrating how you can use the tool
- a reference section describing how to use the capabilities of each tool.

After reading this manual and completing the tutorials, you will know how

- to create and modify source files using the Editor;

- to generate an executable program using the Assembler;
- to test the execution of your programs using the Bugbyter and to fix any errors that you find.

This manual does not teach you how to program in assembly language. It is written for people who are familiar with programming in at least one language (BASIC or Pascal) on an Apple II computer system, and who have read at least one book on 6502 assembly-language programming.

### **What You Need**

To use the assembly-language tools included in this tool kit, you will need

- an Apple II, Apple II Plus, or Apple IIe computer with at least 48K of RAM memory
- a video monitor (The Editor and the Bugbyter do not operate properly if you use an external terminal.)
- at least one disk drive and controller card.

Although it is not required, you may also find that a printer can be very helpful, since you may want to print your assembly listings and use these listings to keep track of your assembly-language programs.

Before you continue reading this manual, you should be familiar with

- how to set up and run your Apple II system (described in the owner's manual that came with your Apple II system)
- how to manipulate disk files using DOS 3.3 (described in the *DOS User's Manual*)
- how to use elementary 6502 assembly-language programming concepts.

You may want to read one of the many guides to assembly-language programming on the Apple II and the 6502 (the microprocessor inside your Apple II) before you read this manual. There are several excellent reference manuals.



---

### ***Guides to Programming the Apple II***

Marvin De Jong, *Apple II Assembly Language*, 1982, Howard Sams & Co., Inc., 4300 W. 62nd St., Indianapolis, IN 46268

A complete manual, with an excellent introductory section, written using an early version of the Apple II Editor/Assembler.

Randy Hyde, *Using 6502 Assembly Language*, 1981, DATAMOST Inc., 19273 Kenya St., Northridge, CA 91326

A thorough manual on the Apple II, including many tables and an introduction to the Apple SWEET16 ROM-coded numeric routines.

Lance Leventhal, *6502 Assembly Language Programming*, 1979, Osborne/McGraw-Hill, 630 Bancroft Way, Berkeley, CA 94710

A quite complete guide to programming the 6502, 6520, and 6522.

---

### ***Reference Manuals on the 6502 Microprocessor***

*Programming Manual, MCS6500 Microcomputer Family*, 1976, MOS Technology, 950 Rittenhouse Rd., Norristown, PA 19401. Pub. #6500-50A

The standard reference for programming the 6502 by the company that designed the 6502 microprocessor.

*R6500 Programming Manual*, 1979, Rockwell International Corp., PO Box 3669, Anaheim, CA 92803

An excellent alternative to the MOS Technology manual; this manual also includes a programming reference card.

*6502 Microprocessor Instant Reference Card*, 1980, Micro Logic Corp., POB 174, Hackensack, NJ 07602. Product #101A

A comprehensive single-card chart of everything you want to know about programming the 6502.

*Applications Information SY6500 Microprocessor Family*, 1980, Synertek Inc., POB 552-MS/34, Santa Clara, CA 95052

An in-depth pamphlet on the internal operation of the 6502 microprocessor, including complete opcode timing diagrams.

# ***Introduction to Assembly-Language Programming***

- 
- 3** The Programming Process
  - 5** Creating an Assembly-Language Source File
  - 5** Assembling Your Program
  - 5** Testing and Verifying Your Program
  - 6** Running Assembly-Language Programs Directly From BASIC
  - 6** Running Assembly-Language Programs From a BASIC Program
  - 6** Advanced Techniques—The Apple IIe Identification Routines
  - 6** A Note on the Tutorials

## ***Introduction to Assembly-Language Programming***

Assembly-language programming is a powerful technique for getting the most out of your Apple II, Apple II Plus, or Apple IIe computer. The programming tools that came with this tool kit will help you write, assemble, and debug assembly-language programs that you can run on any Apple II system. The overview in this chapter explains how you can use these tools to create working assembly-language programs.

---

### ***The Programming Process***

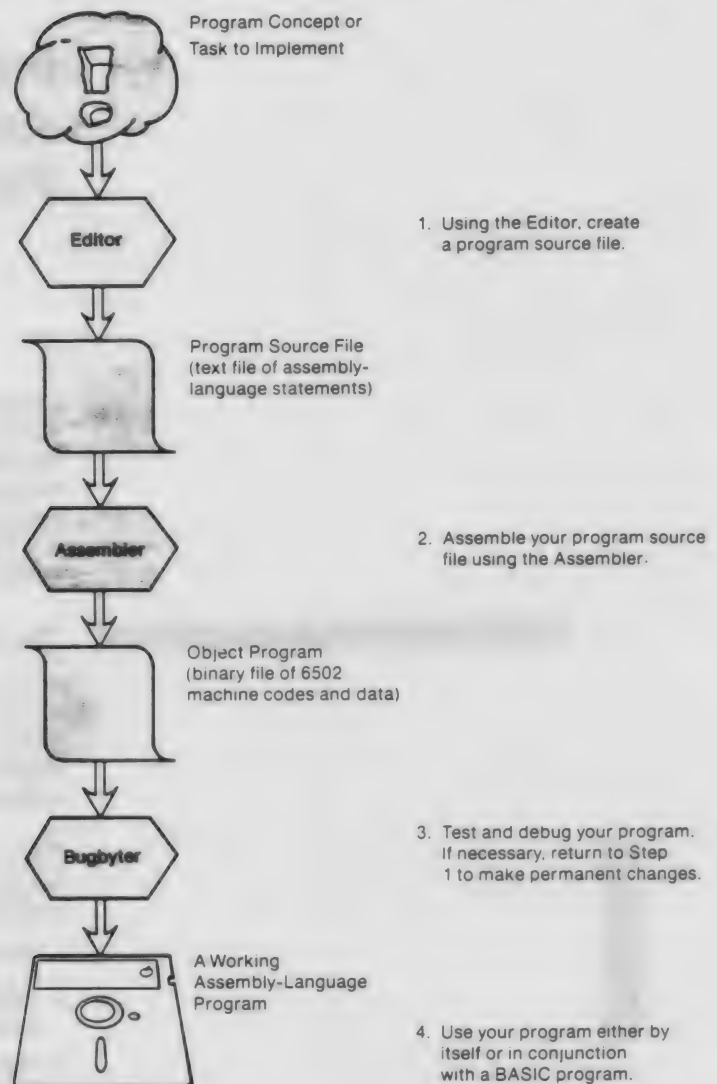
Any program starts as an idea or a task that you want to accomplish. As you organize your thoughts about how your Apple II can accomplish your task, you define the logic of a program. A program is merely an ordered set of instructions designed to accomplish a specific task.

Translating these thoughts into a working Apple II assembly-language program is a process that has several distinct steps. These steps are

1. Creating an assembly-language source file using the Editor;
2. Assembling your source file using the Assembler to create an executable object program;
3. Testing and debugging your program to ensure that it contains no errors;
4. Running your working program, either as a stand-alone program or in conjunction with a BASIC program.

Figure 1-1 illustrates this assembly-language programming process.

**Figure 1-1.** The Process of Assembly-Language Programming



How to use the Editor: see Chapter 2, which also includes a tutorial.

**Assembly-language source files** are files of text characters that represent assembly-language instructions and operands. An **operand** is the address or data that the instruction acts upon.

**Mnemonics** represent individual assembly-language instructions. An **identifier** is a symbolic name that represents an address or a data element.

How to use the Assembler: see Chapter 3, which also contains a tutorial.

How to use Bugbyter: see Chapter 4, which also includes a tutorial.

---

### ***Creating an Assembly-Language Source File***

In translating your thoughts into an actual assembly-language program, you use the Editor included in this tool kit to write assembly-language instructions into a text file and save this text file on a disk.

The text files you create using the Editor that contain assembly-language instructions are called **assembly-language source files**. An assembly-language source file is not an executable program: it is an organized file of text characters that represent assembly-language instructions and operands. In this source file you use a number of three-letter sequences, called **mnemonics**, to represent individual assembly-language instructions. You can represent an address or an element of data in your program either explicitly or by assigning it a symbolic name; you can refer to it by this name. These symbolic names are called symbols or **identifiers**.

---

### ***Assembling Your Program***

To translate your assembly-language source files into executable object code, you use the Assembler that is included in this tool kit. The Assembler is a portion of the combined Editor/Assembler program.

The Assembler translates the instruction mnemonics in your source file into actual machine instruction codes, and translates the identifiers that you used into the actual data or memory references to be used by your computer. The Assembler then stores the assembled program on a disk in the form of a binary file.

---

### ***Testing and Verifying Your Program***

Before you try to run your assembly-language program, you want to make sure that it executes correctly. You can test your program (and fix any errors that you find) using the Bugbyter program included in this tool kit. Using Bugbyter, you can easily step through any portion or portions of your assembly-language program, checking that each instruction executes properly and that the correct data is written to the proper locations. Bugbyter visually shows you every action taken by your Apple II when it executes an assembly-language instruction.

The *DOS User's Manual* describes how to use the BRUN command.

---

### ***Running Assembly-Language Programs Directly From BASIC***

Depending upon your program, the binary files created by the Assembler can form a complete executable program. To run this executable program on your Apple II, you simply use the DOS BRUN command from either Applesoft or Integer BASIC.

Relocating Loader subroutines: see Chapter 5.

---

### ***Running Assembly-Language Programs From a BASIC Program***

You may have written your assembly-language program as a subroutine that you intend to call during the execution of a BASIC program. This combines the advantages of both BASIC and assembly-language programming. To call your assembly-language subroutine from a BASIC program, you can use either the BASIC BLOAD command, or for greater flexibility you can use the Relocating Loader subroutines (RBOOT and RLOAD) that are included in this tool kit.

Identification subroutines: see Chapter 6.

---

### ***Advanced Techniques—The Apple IIe Identification Routines***

When you are writing your assembly-language programs to run on more than one Apple II system, you may want to take advantage of the unique features of the system on which your program is currently running. To do this, your program has to identify the capabilities of that particular Apple II, Apple II Plus, or Apple IIe computer.

To help you to write your programs to do this, two identification subroutines are included in this tool kit. One identification routine is an assembly-language subroutine that you can include directly in your assembly-language source file; the second is written as a BASIC subroutine that you can call as part of a BASIC program.

### ***A Note on the Tutorials***

The tutorials in Chapters 2, 3, and 4 are designed to introduce you to assembly-language programming using the programming tools in this tool kit. In the Editor tutorial, you will create an assembly-language source file and store this file on a disk. In the Assembler tutorial, you will assemble this source file and produce an executable object or binary program. In the Bugbyter tutorial, you will test the operation of this binary program and verify that it executes correctly.



Since each tutorial builds on the results of the earlier tutorials, go through them in sequence so that you will have the proper files ready when you need them. Although you don't have to do the tutorials to understand the tools that are described in this manual, you will probably find the tutorials to be the quickest way to begin using these tools to program in assembly language.

**Before You Start:** Before you use any of the tools in this tool kit, you should make a backup copy of your DOS Programmer's Tool Kit disks. The *DOS User's Manual* that came with your disk drive contains a complete explanation of how to make backup copies of your disks.

## ***The Editor***

---

11	A Tutorial: Editing Text Files
12	Getting Started
14	The Editor Command Level
15	Entering Text With the Editor
16	Displaying Text
17	Line-Editing Text
18	Storing and Retrieving Files
19	Writing a Program With the Editor
21	Leaving the Editor
21	Using the Editor
22	The Editor Command Level
22	Getting Help
23	Typing Uppercase and Lowercase Characters
24	Executing Direct DOS Commands
24	Saving and Retrieving Files on a Disk
25	The Current Disk
26	Loading a File
26	Appending Files
27	Saving Your Edited Files
28	Viewing the Disk Contents
28	Changing the Editor's Current Disk
29	Manipulating Lines in the Text Buffer
29	Adding Lines
30	Inserting Lines
30	Deleting Lines From the Buffer
31	Replacing Lines in the Text Buffer
31	Copying and Moving Lines
32	Clearing the Text Buffer
32	Viewing Your Text in the Text Buffer
32	Listing Lines of Text
33	Relisting Lines of Text
34	Printing Lines of Text
34	Activating a Printer

34	Changing Text Within a Line
35	Searching Text
36	Changing Text
37	Changing the Command Delimiter
38	Entering Edit Mode
38	Character Editing Using Edit Mode
41	Editing Two Files at Once
42	Altering the Display
42	Setting Tabs
43	Using a 40- or 80-Column Display
44	Truncating the Display
44	Leaving the Editor and Returning to BASIC
45	Entering the Monitor
45	Identifying the Absolute Location of Text in Memory

## ***The Editor***

The Editor's text buffer holds more than 26,000 characters. This lets you edit assembly-language source files about 1300 lines long (if an average line contains 20 characters).

A line of text is a sequence of up to 254 characters ending with `RETURN`.

You can use the Editor to edit DOS EXEC files or BASIC program source files. See Appendix F.

Appendix A contains a summary of the Editor commands and functions.

The Editor program in this tool kit is a tool for creating and modifying blocks of text that you can store as DOS 3.3 text files. It lets you manipulate any file of text that fits in the Editor's text buffer. Typically, you use the Editor to create assembly-language source files that you can later use with the Assembler.

You can edit individual characters in your text file, and move or edit whole lines of text. The Editor treats each line of text as a single unit of information: most of the commands you type to manipulate your file will refer to a particular line or group of lines. The Editor keeps track of every line in your file and lets you refer to a particular line using that line's relative position from the start of your file.

This chapter describes how you use the Editor to create and modify text files and to store them on a disk:

- A brief tutorial introduces the Editor, showing you how to create and modify an assembly-language source file. You will use this assembly-language source file later when you work through the tutorials on the Assembler and the Bugbyter debugger.
- The remainder of this chapter contains a detailed description of the functions that you can perform using the Editor.

### ***A Tutorial: Editing Text Files***

This tutorial teaches you the basic editing commands that you need to create assembly-language source files using the Editor. When you finish with this tutorial, you will know how

- to start the Editor program;
- to create a text file by typing lines of text to the Editor;
- to edit or change the lines in your text file using the Editor commands;
- to store your text file on a disk;

- to retrieve your text file from a disk to revise it using the Editor;
- to store your text as an assembly-language source file on your system disk.

### Getting Started

What You Do	What Happens
1. Insert the DOS Programmer's Tool Kit Volume II disk into your disk drive and turn on the power to your Apple II.	When your Apple II has loaded the Applesoft system, you see the Applesoft prompt character ({}).
2. From this Applesoft/DOS command level, type  RUN EDASM	Your Apple II loads the Editor/Assembler system.

Always press (RETURN) after typing an instruction.

and press (RETURN).

When the Editor finishes loading the program, you see this screen display:

```
EDITOR-ASSEMBLER
LOADING EDASM64.3
LOADING EDASM64.3B
LOADING EDASM64.4
LOADING EDASM64.5
LOADING EDASM64.6
```

A 48K Apple II system may show a slightly different screen message.

```
INSERT SOURCE DISKETTE AND PRESS RETURN
```

This message tells you that you may take your system disk out of the disk drive and insert a program source disk if you prefer. Since you will not be using large amounts of disk space during this tutorial, you can leave the system disk in the drive. Be sure that you make a backup copy of your system disk before you proceed.

The **Assembler ID Stamp** is a short text file used to store a serial number and date on each of your source disks: see Chapter 3.

3. Leaving the system disk in your disk drive, press (RETURN).

The Editor reads the current **Assembler ID Stamp** from your system disk and displays it on the screen. The Assembler ID Stamp is a 17-byte ASCII file with the name ASMIDSTAMP that stores a serial number and date stamp on your source disks. The

Assembler uses this ASMIDSTAMP to help you keep track of your assembly-language source listings and disks. Although the Assembler does not check the format of this file, the following format is suggested:

dd-mmm-yy #123456

The Editor allows you to edit these 17 characters to set the current date. There is no required format for these characters; you can type the date any way you wish.

```
EDITOR-ASSEMBLER //
```

```
LOADING EDASM64.2  
LOADING EDASM64.3  
LOADING EDASM64.4  
LOADING EDASM64.5  
LOADING EDASM64.6
```

CURRENT ASSEMBLER ID STAMP IS:

DD-MMM-YY #000000

4. Use the arrow keys to move the cursor, then type in the current date for the Assembler ID Stamp, or simply press **(RETURN)** to accept the ID Stamp as it is shown. Make sure that the cursor is at the right of the Assembler ID Stamp before you press **(RETURN)** to accept all of the characters on the line.

After you press **(RETURN)**, the Editor saves the updated Assembler ID Stamp back onto your source disk. You then see a colon prompt and the cursor in the lower-left corner of the screen. This shows that you are now at the Editor command level.

```
EDITOR-ASSEMBLER //
```

BY JOHN ARKLEY

(C) COPYRIGHT 1982

APPLE COMPUTER INC

:X



## The Editor Command Level

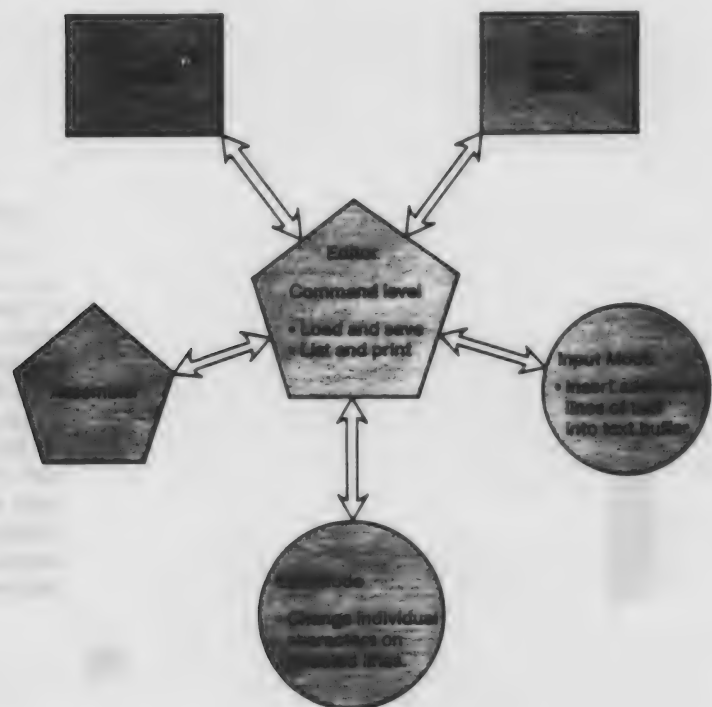
You are in the the Editor command level whenever you see the colon (:) prompt and the cursor at the left margin.

Editor prompt character = colon (:).

**By the Way:** For a 40-column display, the cursor appears as a blinking underline character. If you are using an Apple II system with an 80-column text card, the cursor appears as a solid, nonblinking box.

When you are at the Editor command level, you can type a wide variety of commands to access all of the capabilities of the Editor: you can add information to the text buffer, change information in the text buffer, and write information from the text buffer to a disk; you can also call the Assembler. Figure 2-1 shows a diagram representing all of the features you can access from the Editor command level.

**Figure 2-1.** Editor Features Accessible From the Editor Command Level



The **text buffer** is a memory area to hold your text.

Use the **File command** to see information about your file and the text buffer.

Use the **Add command** to enter Input mode to add lines of text to the text buffer.

Press **(RETURN)** after typing each line of text.

In Input mode, the Editor interprets **(SPACE)** (a space character) as a tab.

To end Input mode, press **(CONTROL)-(Q)**, and press **(RETURN)**.

## Entering Text With the Editor

When you first start the Editor, the Editor creates an area of memory, called the text buffer. This text buffer is empty until you put some characters or text into it.

1. To see the size of the text buffer, type

FILE

and press **(RETURN)**.

The Editor displays three lines of information: the current filename (there is none), the current disk drive, slot, and volume; the number of bytes used in the text buffer; and the number of unused bytes remaining in the text buffer. Your display looks something like this:

```
: FILE
, D1, S6, V 0
0 BYTES USED
26880 BYTES REMAINING
:*
```

2. To add some lines of text to the text buffer, when you see the colon prompt, type

**(A)**

and press **(RETURN)**.

The Editor displays the number 1 to signify that the next line you type will be added to the text buffer as the first line of text.

3. Type the following lines of text, *exactly* as they appear below. After typing each line, press **(RETURN)**.

The Editor accepts the lines of text as you type them and places them into the text buffer.

```
(SPACE)ORG(SPACE)$1000
START(SPACE)
LDY(SPACE)##00
(SPACE)LDX(SPACE)#0
```

4. When you finish typing these lines, press **(RETURN)** after the last line, press **(CONTROL)-(Q)**, and then press **(RETURN)** a second time.

Typing **(CONTROL)-(Q)** and then pressing **(RETURN)** ends the Editor's Input mode and places you back at the Editor command level.

Use the **Print command** to display lines of text from the text buffer. You can also use this command to obtain printed copies of your files.

## Displaying Text

1. To view the lines of text that you just typed into the text buffer, type

(P)

The Editor displays the text that you typed and inserts extra spaces wherever you typed a space character.

and press (RETURN).



```
: P
START          ORG    #1000
                LDY    #$C0
                LDX    #0
:*
```

**Notice:** The Editor interprets space characters in your text as tab characters. The Editor's default tab settings align the text in a reasonable format for assembly-language source files.

Use the **List command** to display lines of text, including relative line numbers.

2. To list your text, including the relative line numbers, type

(L)

The Editor redisplay the lines in the text buffer:

and press (RETURN).



```
: L
1 START          ORG    #1000
2                LDY    #$C0
3                LDX    #0
:*
```

A relative line number is simply the position of each line of text relative to the beginning of the text buffer; line number two is always the second line of your file, and so on. Relative line numbers have nothing to do with any characters or numbers stored in your text file. This means that if you insert or delete lines of text within your file, the relative position of all subsequent lines of text will change.

A **relative line number** is the position of a text line relative to the beginning of the text buffer; these numbers change as lines of text are inserted and deleted.

**Relative Line Numbers:** The Editor's relative line numbers should not be confused with statement numbers you might have used in BASIC programs. These BASIC statement numbers are actual numbers that you stored as part of the program text. If you are using the Editor to edit a BASIC program, you see two numbers associated with each line:

- the relative line number calculated by the Editor
- the statement number that is part of every BASIC statement line.

Use the **Edit command** to place the Editor in Edit mode to allow you to modify the text in the text buffer.

---

### Line-Editing Text

When you make typing mistakes, you can easily fix them by using the Edit command.

1. To edit each line of your text in turn and fix any typing errors, type

(E)

and press (RETURN).

The Editor displays the first line of your text on the screen, with an inverse-video cursor positioned over the first character of the line.



1

⌘ ORG #1000

---

**Remember:** The first character that you typed was (SPACE). Since it is not a visible character, the cursor doesn't appear in inverse video. Use (→) to move the cursor to the next character to see the cursor in inverse video.

You can now edit the contents of this edit line by using the following procedure:

- To move the cursor along the line, use (→) and (←).
- To replace a character, type the new character over the old character.
- To delete a character, place the cursor over that character and press (CONTROL)-(D).
- To insert a character, place the cursor over the character that will follow the character you are going to insert, and then press (CONTROL)-(I). Type the characters you want to insert. When you are done, press (→) or (←) to end the insertion.
- To accept the edited line as it appears on the screen, press (RETURN).

---

2. Follow the procedure above to edit each line of your text until it appears exactly as shown in the previous sections. When no errors remain, press (RETURN) until you are back at the Editor command level.

As you press (RETURN) after each line, the Editor incorporates all the changes you made to the line displayed on the screen, and then displays the next line for you to edit.

---

Use the **Save command** to save the text buffer in a disk file.

The Editor's **CAT command** is simply a shortened version of the DOS **CATALOG** command.

The Editor creates only **text files**.

---

### Storing and Retrieving Files

When you are back at the Editor command level, you can save your text on the system disk.

- 
1. To save your text, type

SAVE TESTPROGRAM

and press **(RETURN)**.

The Editor saves your text in a text file using the name TESTPROGRAM.

- 
2. To verify that your text file was created correctly, type

CAT

and press **(RETURN)**. You may have to press **(RETURN)** a second time to see all of the files on the disk.

The Editor displays the contents of your disk, with the text file TESTPROGRAM shown at the end of the other files.

:CAT

DISK VOLUME 100

A 004 HELLO  
B 012 .  
B

T 002 TESTPROGRAM

:X

---

Note that your file has the symbol T for text shown next to the TESTPROGRAM filename in the catalog. The Editor creates only text files; there is no difference between your assembly-language program and any other type of text file.

Use the **New** command to clear the text buffer.

3. To show that your text is still in the Editor's text buffer, type

(L)

and press (RETURN).

The Editor displays the contents of the text buffer, showing that your text is still there as well as on your disk.

4. To clear the text buffer, type

NEW

and press (RETURN).

The Editor clears the text buffer, so you can type in new text or perform some other function. If you type (L) and press (RETURN), the Editor shows you that there is nothing in the buffer.

5. To reload into the Apple II's memory the text file that you just saved onto your disk, type

LOAD TESTPROGRAM

and press (RETURN).

The Editor loads your text file from the disk into the text buffer.

Use the **Load** command to load a text file from a disk into the text buffer.

6. To see that your text is now back in the text buffer, type

(L)

and press (RETURN).

The Editor displays the text on your screen, exactly as it looked before you saved it.

### Writing a Program With the Editor

You have written the first three lines of an assembly-language source file that you now have in the text buffer. To finish this program, you will add more lines to the text buffer and then save the final version on a disk.

1. To begin adding more lines, type

(A)

and press (RETURN).

Notice that the relative line number 4 appears before the prompt. You are now ready to enter the rest of your program.



2. Type the following lines. After each line, before pressing **(RETURN)**, make sure that the line is typed exactly as it appears below, including spaces.

```

LOOP (SPACE)JSR (SPACE)STORE
(SPACE)INX
(SPACE)CPX (SPACE)#5
(SPACE)BNE (SPACE)LOOP
(SPACE)RTS
STORE (SPACE)INY
(SPACE)TYA
(SPACE)STA (SPACE)BUFF,X
(SPACE)RTS
(SPACE)DS (SPACE)$ED,$00
BUFF (SPACE)DS (SPACE)10,$00

```

**By the Way:** If you want to return to the Editor command level when you finish typing, press **(CONTROL)-(Q)** and then press **(RETURN)**.

3. To List what you've typed, type

**(L)**

The Editor displays the text lines that you typed. Your program should appear exactly as shown:

and press **(RETURN)**.

1		ORG	\$1000
2	START	LDY	#\$C0
3		LDX	#0
4	LOOP	JSR	STORE
5		INX	
6		CPX	#5
7		BNE	LOOP
8		RTS	
9	STORE	INY	
10		TYA	
11		STA	BUFF,X
12		RTS	
13		DS	\$ED,\$00
14	BUFF	DS	10,\$00

4. If there are any mistakes, use the Edit command you just learned to correct these errors.

---

5. When you have finished, type `SAVE` and press `(RETURN)`. The Editor saves your completed assembly-language program on your disk, using the name `TESTPROGRAM` that you used to load your file.

---

---

### **Leaving the Editor**

---

Use the **End** command to leave the Editor and return to Applesoft BASIC.

1. To leave the Editor and return to BASIC, type `END` and press `(RETURN)`. You see the Applesoft prompt `( )`.

---

When you type `END`, if you are using an 80-column text card in your Apple II, your video display reverts to 40 columns.

**Do It Again:** If you want to practice the tutorial some more, when you see the Applesoft prompt, type `MAXFILES 5` and press `(RETURN)`. Then type `CALL 3075` and press `(RETURN)` again. You are back at the Editor command level when you see the colon prompt.

You have now completed the tutorial. In it, you learned how to use the Editor to do a variety of tasks. You now know how

- to start up the Editor;
- to enter text into the Editor's text buffer;
- to edit text in the text buffer;
- to view your text that is in the Editor's text buffer;
- to save your text into a text file on a disk;
- to retrieve your text file from the disk to edit and then save it again.

You will use the assembly-language source file you just created in the tutorials on the Assembler and the Bugbyter debugging program. Do not delete this text file from your disk if you want to do these other tutorials.

---

### **Using the Editor**

---

The preceding tutorial introduced you to some of the basic capabilities of the Editor. In the sections that follow, all these capabilities are described in more detail, plus other capabilities that are not included in the tutorial.

Lowercase letters shown in a command name indicate *optional characters* that you need not type.

The colon (:) is the Editor's **command delimiter**.

To get help, type a ? to view the Editor's reference card.

---

### **The Editor Command Level**

You are at the Editor command level whenever you see the colon (:) prompt character, followed by the cursor, at the left margin. You can access all the Editor features by typing commands from this level.

When you type these commands, you can often abbreviate the command names to just one or two letters. In the remainder of this chapter, command names are shown with the required letters in uppercase and any optional letters in lowercase. When you type the command, you can type in either uppercase or lowercase, and include as many of the optional characters as you wish. The Editor ignores any spaces that you insert before or between the command and its parameters.

You can type more than one command on a line. This allows you to do several editing operations at once. To separate each command on the line, you must type the Editor command delimiter, a colon.



#### **Warning**

To avoid confusion and any unintentional loss of the text in your text buffer, be careful when typing more than one command on the command line. Whenever you type the Add, Delete, Insert, Copy, or Replace commands, the relative line numbers of all subsequent text lines may be changed. The Editor uses these new relative line numbers when executing any subsequent commands. The Editor performs each command in turn, unless an error occurs while performing one of them, or unless you call the Assembler. A failure to locate the target of a character or string search is not considered an error.

To ensure that you do not unintentionally delete text from your file, always check the relative line number of any text line before you replace or delete it. Do not type more than one command on the command line when using the Replace or Delete commands.

---

### **Getting Help**

When you are first learning to use the Editor, you may find it difficult to remember all the Editor commands and syntax. The Editor gives you a *reference card* to help remind you of these commands and their syntax.

To view this aid, type a question mark

?

following the colon prompt and press **(RETURN)**. The Editor displays a table on the screen showing all the Editor commands and the common syntax for each. This table is longer than a full screen; to see the rest of the table, press **(SPACE)** or **(RETURN)** or any character key.

---

### Typing Uppercase and Lowercase Characters

If you want to insert lowercase characters in your text (for example, you may want your program to display a message in uppercase and lowercase), type lowercase characters using the most convenient method for your particular Apple II system.

- If you have an Apple IIe computer, or if you have an Apple II or Apple II Plus computer with the one-wire shift-key modification installed, and an ALS Smarterm 80-column display card in slot 3: enter lowercase characters using your Apple II's lowercase input facilities.
- If you have an Apple II or Apple II Plus computer, with the one-wire shift-key modification installed, but you do not have the ALS Smarterm 80-column display card: activate your keyboard shift-key by typing

**SET LCase**

and pressing **(RETURN)**.

After you turn on this lowercase character capability, you can return to uppercase character entry by typing

**SET UCase**

and pressing **(RETURN)**. You cannot type lowercase characters until you again type a Set Lowercase command.

- If your Apple II or Apple II Plus computer does not have a keyboard shift-key modification to allow you to type lowercase characters, you can use the Editor's control-character **(CAPS LOCK)** command to act as a software shift key. When you first start the Editor, the Editor sets the keyboard in **caps lock mode**. To unlock the keyboard and enable lowercase character entry, press **(CONTROL)-(E)** and then press **(RETURN)**. To reenter caps lock mode, press **(CONTROL)-(W)** and then **(RETURN)**.

**Remember:** Whenever you are entering lowercase characters, you should remember that, although the Editor recognizes commands typed in either uppercase or lowercase, filenames and character strings are all sensitive to case. This means that *EditA* and *edita* are recognized as two different filenames under DOS 3.3 and are seen as two different strings when the Editor is searching following a Find, Change, or Edit command.

The **Set Lowercase command** activates the keyboard shift-key for the Apple II or Apple II Plus.

The **Set Uppercase command** deactivates your keyboard shift-key after you have activated it with the Set Lowercase command.

DOS commands must be preceded by a period (.).

---

### **Executing Direct DOS Commands**

Any time you are using the Editor, you can perform regular DOS file-handling functions without leaving the Editor command level. This means you can manipulate disk files using the familiar DOS file commands. Only use direct DOS commands from the Editor to perform the following functions:

- Rename a file (RENAME)
- Delete a file (DELETE)
- Lock a file (LOCK)
- Unlock a file (UNLOCK)
- Obtain a file catalog (CATALOG)
- Turn on the Monitor (MON)
- Disable the MON command (NOMON).

To execute a DOS command from the Editor command level, type a period (.) as the first character on the Editor command line, followed by the DOS command exactly as you would type the command in Applesoft or Integer BASIC.

For example, if you type the command

```
.RENAME PHONE LIST.DIRECTORY.D2
```

you command DOS to rename a file on a disk in drive 2. You must supply the slot and drive parameters when you type a direct DOS command; direct DOS commands do not use nor do they alter the Editor's current slot and drive settings.



---

### **Warning**

Using any DOS commands other than RENAME, DELETE, LOCK, UNLOCK, CATALOG, MON, and NOMON from within the Editor may destroy your text in the Editor's text buffer. For this reason, you should be very careful when using any direct DOS commands to avoid typing a DOS command that will destroy any text in memory.

---

The Editor does not work on random-access text files. To edit BASIC programs with the Editor, see Appendix F.

---

### **Saving and Retrieving Files on a Disk**

The Editor allows you to save and retrieve text files to and from a disk. The Editor can work with any DOS sequential text file, and is not limited to assembly-language program source files. Thus the Editor can be used to create and modify EXEC files and to examine and modify sequential data files written by BASIC programs, if these files will fit in the Editor's buffer space.

Volume 0 is a **wildcard** representing any volume.

### **The Current Disk**

The Editor remembers the last disk slot, drive, and volume that you referenced when you are using the Editor, much as does DOS. When you load a file from a disk or save a file to a disk, the Editor sets its *current* disk slot, drive, and volume to refer to the disk slot, drive, and volume that you just used. When you first enter the Editor, the Editor's current settings are the boot slot, drive 1, and Volume 0.

The Editor also remembers the filename that you used when you last loaded or saved your file. This filename is called the **current filename**, and is used by the Editor to save you from having to retype the filename every time you want to back up your file.

To view the Editor's current filename, slot, drive, and volume, type

FILE

and press **(RETURN)**. Use the File command to display the current file name, slot, drive, and volume information, plus the current length of the text and the amount of space (in bytes) remaining in the buffer.

If you have not yet executed a Load or Save command during an edit session, there is no current filename associated with your text. The Editor will display the slot, drive, volume, and length information in any case.

To learn the amount of memory used by your text and the amount of unused memory, type

LENGTH

and press **(RETURN)**. Use the Length command to display the size of the text in the current text buffer and the amount of memory remaining, in bytes (characters). The sum of these two numbers is the total memory available to the Editor at the current time.

In this manual square brackets [] indicate optional command parameters.

When you type a file command to the Editor, you can specify a slot, drive, and volume after the filename just as you do under DOS. If you do not specify a slot, drive, or volume, the Editor searches for the file using the Editor's current slot, drive, and volume settings. In the discussions that follow, square brackets [] are used to indicate optional command parameters.



Use the **Load command** to load a text file from a disk into the text buffer.

### **Loading a File**

To load a text file that is stored on a disk, type

```
LOAD filename [ ,Sn] [ ,On] [ ,Wn]
```

and press **(RETURN)**. The Editor loads the text file that you specify into the Editor's text buffer. Any text previously in the buffer is erased. After the Editor loads the text file, you can edit the file using any of the editing commands.

Error messages: see Appendix D.

If the Editor does not find the file you specified, the Editor displays the familiar DOS FILE NOT FOUND message. Other error messages that may appear are discussed in the Appendixes.

### **Appending Files**

Use the **Append command** to append a text file from a disk to the end of your text in the text buffer.

You can put two or more text files together by using the Editor Append command. To copy a text file onto the end of the text that you are currently editing, type

```
APPEND filename [ ,Sn] [ ,On] [ ,Wn]
```

and press **(RETURN)**. The Editor loads the specified text file from the disk and adds it to the text buffer at the end of your existing text. This command does not affect the Editor's current filename, which remains as the filename used in the previous Load command.

To insert a file of text into the middle of the text you are editing, you must type three commands:

- First append the text file to the end of your existing text using the Append command.
- Then use the Copy and Delete commands to move the text wherever you want within the text buffer. The Copy and Delete commands are discussed later.

You can append a text file after a particular line in your program, replacing any text that currently follows this line with the text from the appended file. To do this, type

```
APPEND line# filename [ ,Sn] [ ,On] [ ,Wn]
```

and press **(RETURN)**. The Editor first deletes all the lines in the text buffer following the line number (line#) that you specify and then loads the specified text file into the text buffer following your existing text.



Use the **Save** command to store the contents of the text buffer in a disk file.

### ***Saving Your Edited Files***

When you finish editing your text, you should save your file on a disk. As you are editing your text, you should also save it occasionally. To save the contents of the text buffer, type

```
SAVE [filename] [,Sn] [,Dn] [,Vn]
```

and press **(RETURN)**. This Save command lets you write the contents of the text buffer into a text file using the filename, slot, drive, and volume that you specify.

If you do not specify a filename, or if you don't specify a slot, drive, or volume, the Editor will save the text buffer using its current filename, slot, drive, or volume settings. For example, if you are saving a file with the same filename you used with the most recent Load command, you need only type

```
SAVE
```

and the Editor saves the text buffer, replacing the original file on the disk.

File command: see the section "The Current Disk."

If you type a Save command without previously specifying a filename with a Load or Save command, the Editor displays the current slot, drive, volume, and length information. At any time, you can find the filename that you used with the most recent Load or Save command by using the File command.

If the disk you specify is write-protected, or if no space remains for your file, the Editor displays the DOS error message. Other error messages may appear: see Appendix D.

### **Warning**

Typing the Save command causes the Editor to write over any file on the selected disk that has the same name as the file you are saving. The Editor does not prompt you when this occurs. To protect your other files from accidentally being overwritten, you should use the direct DOS `.LOCK filename` command to write protect the text files on a disk when you use the Editor/Assembler. If you have used the Lock command to lock all the text files on a disk and only unlock the file you wish to edit, you cannot accidentally overwrite a file if you should misspell a filename when typing the Save command. (This can easily happen when you are working with multiple source files whose names differ only slightly.)

When the Editor overwrites a file with the same name as the text you are saving, the Editor does not delete the original file before overwriting it. This means that if the text you are saving is shorter than the original file on disk, the disk space allocated for the original file is not released for other uses. To conserve disk space, type a direct DOS `.DELETE filename` command before using the Save command.

Do not issue a direct DOS `.SAVE` command in place of the Editor Save command. The DOS command causes DOS to *save the contents of the current BASIC program*. Since there is no such program, this command has unpredictable results.

**Be Careful:** While editing, *back up your work* every twenty minutes or so by saving the text buffer to your disk. Don't get so busy editing that you forget. Although the chances of accidentally losing data are slight, a one-second loss of power to the computer can wipe out hours of editing. It is possible to lose your only copy of a file, or an entire disk, if a power loss occurs while you are saving to your disk. You can protect yourself against this type of loss by alternately saving your program to at least two disks.

#### **Viewing the Disk Contents**

Whenever you are using the Editor, you can view the contents of a disk by typing

```
CATalog C,Sn] C,Dn] C,Vn]
```

and pressing **(RETURN)**. This is equivalent to typing the DOS **.CATALOG** command

```
.CATALOG C,Sn] C,Dn] C,Vn]
```

The Editor's **CAT** command is a short form of the DOS **CATALOG** command.

The main difference is that you can abbreviate the Editor command to the shorter **CAT** to save typing. The direct DOS **.CATALOG** command lets you catalog disks on other drives without altering the Editor's current slot, drive, and volume settings.

#### **Changing the Editor's Current Disk**

The Editor has three commands that let you change the Editor's current slot, drive, and volume settings. However, since you can just as easily specify a slot, drive, and/or volume when you type the Load, Append, Save, or Cat commands, you probably will not use these three commands often.

To change the currently-selected disk slot, type

```
Slot slot#
```

and press **(RETURN)**. The Slot command selects the slot number that the Editor will use in subsequent file Load, Append, Save, and Cat commands. The slot number (slot#) must be a number between 1 and 7 and this slot must contain a disk controller card.

To change the currently-selected disk drive, type

```
Drive drive#
```

and press **(RETURN)**. The Drive command selects the current disk drive from which Editor will read or write text files. The drive number (drive#) must be either 1 or 2; if you select any other value, the Editor displays an error message.

To change the currently-selected disk volume, type

Vol vol#

and press **(RETURN)**. The Volume command selects the volume number of the disk from which the Editor will read or write text files. A volume number is required only when your system has a large number of volumes and uses the **virtual volume** mechanism that is typically used with large disk storage devices. If you are using only Disk II drives, you need not specify a volume number to read from or write to a file on a disk.

---

### ***Manipulating Lines in the Text Buffer***

The Editor accepts a number of commands to support adding, deleting, copying, inserting, and replacing lines within the text buffer.

#### ***Adding Lines***

Use the **Add command** to enter Input mode so that you can add lines of text to the text buffer.

To add lines to the text buffer, type

Add [line#]

and press **(RETURN)**.

You use the Add command to add new lines to the end of the text buffer or to add lines after a specified line number. If you type **A**, **AD**, or **ADD** and do not specify a line number, the Editor displays the number of the next line to be added to the end of the buffer and places you in Input mode.

When you are in Input mode, the Editor accepts anything you type and inserts it into the text buffer. You can type any number of lines into the buffer, terminating each line by pressing **(RETURN)**. After you press each **(RETURN)**, the next line number is displayed.

After you type your last line of text, press **(RETURN)** to enter this line and then press either **(CONTROL)-(D)** or **(CONTROL)-(Q)**. Press **(RETURN)** immediately after the next line number appears on the screen. This terminates Input mode and returns you to the Editor Command level.

You can also use the Add command with a line number. When you do this, the Editor places you in Input mode, but any lines that you type are added just *after* the line with the relative line number that you specified. (The Insert command places any lines that you type *before* the line with the specified relative line number.)

**Reminder:** In Input mode you can use all the normal input editing functions, including the arrow keys. You can type uppercase and lowercase characters, using the facilities of your particular Apple II. If your Apple II has the Autostart ROM, you can also use the additional **(ESCAPE)** features for cursor movement.

Input mode allows you to place control characters into your text file. Be careful not to insert control characters into your assembly-language programs, because the Assembler does not accept control characters in assembly-language source files. When you are using Input mode, control characters in your text are not displayed. They are displayed in inverse video when you use the List command to list your text.

#### **Inserting Lines**

The **insert command** allows you to insert lines of text between existing text in the text buffer.

To insert lines in the text buffer before a particular line number, type

Insert *line#*

and press **(RETURN)**. Typing the Insert command places you in Input mode. Any lines that you type following this command are inserted into the text buffer just *before* the line with the relative line number that you specify. Typing **INSERT 1** inserts lines before the first line in your text file. Typing **INSERT 0** or **INSERT *line#***, where *line#* is higher than the last line number in the text buffer, works the same as the Add command: the text that you type is added to the end of the buffer.

**(CONTROL)-(D)** or **(CONTROL)-(Q)** terminates input mode.

You can terminate Input mode by pressing either **(CONTROL)-(D)** or **(CONTROL)-(Q)**. Remember that when you insert lines into your text, the relative line numbers of all lines that follow these lines in the text buffer will be increased. Check the new relative line numbers before you do any further editing to these lines.

#### **Deleting Lines From the Buffer**

The **Delete command** lets you delete lines of text from the text buffer.

To delete lines from the text buffer type

Delete *begin#* [-*end#*]

and press **(RETURN)**. Typing the Delete command deletes all the lines starting with the line numbered (*begin#*) and ending with the line numbered (*end#*). The beginning line number must be separated from the ending line number by a dash (-). If you do not type a dash and an ending line number, the Editor deletes only the line numbered (*begin#*).

**Remember:** Each time you use the Delete command, the relative line numbers of all subsequent lines in the text buffer are decreased.

### ***Replacing Lines in the Text Buffer***

The **Replace** command lets you replace a range of lines with new text.

To replace several lines of text in the text buffer, type

Replace begin# [-end#]

and press **(RETURN)**. Typing the Replace command is the same as typing a Delete command followed by an Insert command starting at the first line number (begin#) that you specified. The Editor first deletes the lines in the range from (begin#) to (end#) and then enters Input mode, allowing you to insert any number of lines to replace those that you deleted. You can insert lines exactly as you would after typing an Insert command. You must terminate input with **(CONTROL)-(Q)** or **(CONTROL)-(D)**.

### ***Copying and Moving Lines***

The **Copy** command lets you copy a range of lines from one location in the text buffer to another.

To copy lines from one part of the text buffer to another, type

Copy begin# [-end#] TO dest#

and press **(RETURN)**. When you use the Copy command, the Editor copies the lines from (begin#) to (end#) and inserts them just before the line numbered (dest#) in the file. If you do not specify (end#), only the line numbered (begin#) is copied. When you type an ending line number (end#), it must be greater than or equal to (begin#). You must type the word **TO** before typing the destination line number, which must be outside the (begin#-end#) range.

To move lines from one part of the text buffer to another, you must use two commands. First, use the Copy command to move the original lines to their new location. Then use the Delete command to remove the lines from their old location.

**Renumbered Lines:** Remember that when you copy lines to a new location, the relative line numbers of all text lines following that location will change. This means that when you copy lines to a (dest#) less than (begin#), the relative line numbers of the original lines will change when the copied lines are inserted before them. Before you delete the original lines, check their new relative line numbers because they change also.

Use the **New** command to clear the text buffer.

### **Clearing the Text Buffer**

To clear all of your text from the text buffer, type

**NEW**

and press **(RETURN)**. Typing this command lets you clear the current text buffer, in effect deleting all of your text from the buffer. You use this command to clear the buffer before typing a new text file or in conjunction with the Swap command.

If you want to save the text in the text buffer, use the Editor's Save command. When you type the New command, the Editor does not prompt you to save your work before it clears the text buffer.

**To Recover:** If you accidentally clear a text buffer, you can possibly restore the buffer by using the Monitor commands. The New command merely resets an end-of-text pointer and does not destroy the contents of the buffer. Using the Monitor, you may be able to determine the previous end-of-text address and then set the end-of-text pointer to this address. The three text-buffer pointers in the Editor are defined in Appendix H.

Test-buffer pointers: see Appendix H.

---

### **Viewing Your Text in the Text Buffer**

The Editor lets you view the text in the text buffer in two different ways:

- with the relative line numbers normally displayed, and the control characters displayed in inverse video. (Use the List command.)
- without relative line numbers, and the control characters sent as control characters to the video display or to a printer. (Use the Print command.)

#### **Listing Lines of Text**

To display lines of text from the text buffer, type

**List [begin# [-end#]]**

and press **(RETURN)**. The List command lets you display lines of text on the screen, and shows the relative line number before each line.

- If you do not specify any line numbers with the List command, the Editor lists the contents of the entire text buffer.
- If you specify a beginning line number (begin#), the Editor begins listing from that line. Otherwise the Editor starts listing lines from the beginning of the text buffer. If the beginning line number is the only number that you specify, the Editor lists only that line.

Use the **List** command to display lines of text with relative line numbers.



- If you specify a beginning line number, a hyphen, and an ending line number (end#) larger than the beginning line number, the Editor lists only those lines in the range from the beginning line number to the ending line number.
- If you specify a beginning line number, a hyphen, and the number of lines you want to list (line count), where the count is smaller than the beginning line number, the Editor lists the counted number of lines starting with the beginning line number.
- If you specify a beginning line number, a hyphen, and no ending line number or line count, the Editor lists all lines from the beginning line number to the end of the text buffer.
- If you type more than one range of line numbers, separating each range with a comma, the Editor lists each range of lines and shows a blank line between each group of lines.

You can interrupt the listing at any point by pressing **(SPACE)**. To continue the listing, press any other key, except **(CONTROL)-(C)**. If you press **(SPACE)** again, the Editor displays one line and stops. This allows you to step through the text buffer, examining one line at a time. You can cancel the listing at any time by pressing **(CONTROL)-(C)**; the Editor will return to the command level.

The Editor displays control characters as inverse-video characters on the screen when you use the List command. If your Apple II system has lowercase display capabilities, lowercase characters display in lowercase. If your Apple II system does not have this capability, lowercase characters will appear as punctuation symbols on the screen.

#### ***Relisting Lines of Text***

Press **(CONTROL)-(R)** to relist the lines you displayed with the previous List command.

The Editor always remembers the line numbers that you specified with the most recent List command. To relist the lines displayed by the previous List command, press

**(CONTROL)-(R)**

and press **(RETURN)**. This lets you display the lines of text as if you completely retyped the previous List command using the same relative line numbers. If you have added or removed lines of text, the Editor may not list the text that you intended.



Use the **Print command** to display lines of text without showing line numbers. This command can also be used to obtain a printed copy of your file.

### **Printing Lines of Text**

To display or print lines of text without relative line numbers, type

```
Print [begin# [-end#]]
```

and press **(RETURN)**. Typing this command displays the indicated text without showing line numbers. This command differs from the List command in two respects:

- Relative line numbers are not printed. The first character of each line is printed in column 1.
- The control characters that the Editor displays in inverse video when you use the List command are sent to the screen or printer as control characters. This lets you insert control characters in a file to activate any printer-control features that you may have in your printer.

### **Activating a Printer**

The DOS **.PR# command** can be used to turn a printer on or off from within the Editor.

To activate a printer from the Editor, you must use the direct DOS command facility: for example, if your system uses a printer interface card, type

```
.PR# slot#
```

and press **(RETURN)**. (slot#) is the number of the slot containing the printer interface card. Once you activate your printer, you can use the Editor List or Print commands to print the current contents of the text buffer.

### **Warning**

Never specify .PR# 6. Slot 6 contains the disk controller card.

### **Changing Text Within a Line**

This section describes the commands you use to find and revise text lines that are already in the text buffer. These commands are Find, Change, and Edit.

Use the **Find command** to search the text buffer for occurrences of particular words or character strings.

### Searching Text

To search through the text for the occurrence of a particular word or string, type

Find [begin#[C-end#]] .<string>.

and press **RETURN**. <string> is the word or character string that you want to search for. This search string must be enclosed by two delimiting characters that do not occur in the string itself. The delimiters are shown above as period characters. In place of the periods, you can also use quotes, slashes, or any punctuation character other than a dash or a comma.

**Notice:** Characters enclosed within brackets, such as <charstr> indicate a character string. When you type a string, do *not* type the brackets.

The Find command lets you locate and list all lines containing the specified search string that fall in the range of relative line numbers from (begin#) to (end#). If you do not specify beginning and ending line numbers, the Editor searches the entire text buffer. The Editor lists a line only once regardless of how many times the string occurs within that line.

**CONTROL**-**A** in a search string is a **wildcard** that matches any character.

You can specify a **wildcard** character within the search string by pressing **CONTROL**-**A**. A **CONTROL**-**A** in the string represents any character, allowing you to search for all sets of similar character strings that may differ in one or more characters.

For example, if you type the command

F .TEST#.

where the # represents a wildcard, and then press **RETURN**, the Editor searches your entire text and lists any lines where the character sequence **TEST** is followed by another character. Any lines containing **TEST1**, **TESTX**, or other **TEST\*** sequences are found and listed.

**Remember:** In this Editor when you press **SPACE**, it will print as a *tab*. This is convenient when you are editing a source program and searching for a specific text line. Instead of the space character, you can use **CONTROL**-**A** (the wildcard character) for this purpose.

Use the **Change** command to replace occurrences of a word or string with another word or string.

### Changing Text

To substitute a new character string for some or all occurrences of an old character string in the text buffer, type the command

Change [begin#[-end#]] .<oldstr>.<newstr>.

and press **RETURN**. The Change command lets you search the text buffer for occurrences of the search string <oldstr> and replace (at your option) any occurrence that it finds with the new string <newstr>. You can use any punctuation character except a dash or a comma as the delimiter instead of the period shown above.

If you want to search a range of lines and you specify a beginning line number (begin#) other than the first line of the text buffer, the search begins from that line. If you type a dash and an ending line number (end#) after the beginning line number, the Editor searches the lines between the beginning line number and the ending line number. If you omit the dash, then the Editor only searches the line with the relative line number (begin#).

Whenever you type the Change command, this question appears

ALL OR SOME?(A/S)?

You can indicate that you wish to change all occurrences of the search string in the lines that you specified (type **A**), or that you wish to change only some of these occurrences (type **S**). If you type any other response, the Editor assumes that you wish to change all occurrences of the search string.

If you respond with **S**, the Editor searches for the first occurrence of the search string. When an occurrence is found, the Editor displays what the text line will look like *after* the change is made, and asks you either to accept or reject the change by displaying

Y/N

If you respond by typing either uppercase or lowercase **Y**, the Editor replaces the original line in the file with the displayed line and continues searching for another occurrence. If you type uppercase or lowercase **N**, the Editor does not make that specific change, but begins searching for the next occurrence of the string. Typing **CONTROL-C** rejects the change, cancels all further changes, and returns you to the command level. You must respond with only these characters; if you type any other character, it is rejected with an audible "beep."

**By the Way:** If you reject a specific change by typing either uppercase or lowercase **(N)**, the Editor does not redisplay the text line in its original unchanged form. The actual text in the text buffer remains unchanged.

When you specify the search string **<oldstr>**, it must contain at least one character. You can use an empty string **<newstr>** to remove occurrences of text from your file. For example, the command **C . JUNK . .** removes all occurrences of the string JUNK from the text buffer (that is, replaces these occurrences with nothing).

You can omit the trailing delimiter following the new string, since pressing **(RETURN)** signals an end of the string. You cannot use the command delimiter as a delimiter in your search strings.

As with the Find command, you can use the wildcard character **(CONTROL)-(A)** in your search string to represent any character. Using the wildcard, you can change a set of similar character strings into the same identical new string. If you type **(CONTROL)-(A)** in the new string **<newstr>**, it is simply entered into your text as a **(CONTROL)-(A)** character.

#### **Changing the Command Delimiter**

The Editor recognizes the colon as a command delimiter to separate commands. You cannot use a colon in a search string, unless you first change the Editor's command delimiter to some other character. Since the colon is not commonly used in assembly-language programs, you probably will not have to change the command delimiter often when you are editing your source programs.

To change the Editor's command delimiter, type

```
SET Delim . <new cmd delimiter> .
```

and press **(RETURN)**. The Editor's command delimiter becomes the character that you enclose within the delimiters (periods) in the command above.

A command delimiter cannot be a space character. If you set the new command delimiter to a space character or simply type **SET** and press **(RETURN)**, the system just prints the current command delimiter and returns to the command level. Do not use a period as the command delimiter, since this is the Direct DOS Command character. The Editor accepts any other printing character (except **(RETURN)**) as the command delimiter.

Use the **Edit** command to enter edit mode; you can then change any characters in the text buffer.

### **Entering Edit Mode**

The Change command is useful for making identical changes to several different lines of text. To make individual changes that will be different for each line, you will want to use the Editor's Edit mode. For example, if you entered some assembler source lines without comments, you might want to go back and add individual comments to a group of lines or to all the lines containing a specific identifier.

To enter Edit mode, type

```
Edit [begin# [-end#]] [.<string>.]
```

and press **RETURN**. When you type the Edit command, you enter Edit mode and see on the screen the first line that you specify.

When you type the Edit command, you must specify the lines that you want to edit:

- If you omit both the optional line numbers and the optional search string, you can edit *all* the lines in the text buffer.
- If you specify a line number or a range of line numbers (begin#-end#), you can edit each of the lines in this range, one after another.
- If you type both a range of line numbers and a search string, or just a search string, you can edit only those lines in the range of line numbers that contain an occurrence of the search string. You can also use the wildcard character (**CONTROL**-**A**) in the search string.

### **Character Editing Using Edit Mode**

Once you are in Edit Mode, you see the next line that you can edit, with the inverse-video cursor on the first character in the line. Using the arrow keys, you can move the cursor to the left or right along the line. Using other control characters, you can replace, insert, or delete characters from the line and immediately see your changes on the screen. The resulting line may be either shorter or longer than the original line.

After you use the arrow keys to place the inverse-video cursor over a character that you wish to edit, you can

- replace the character by typing a new character over the old character;
- delete the character by pressing **CONTROL**-**D**;

In Edit mode use **CONTROL-V** to enter a control character. Control characters are displayed in inverse video.

- insert additional characters before the current character by pressing **CONTROL-I** and typing the additional characters. End the insertion by pressing an arrow key, any control character other than **CONTROL-V**, or pressing **RETURN**.

If you want to enter a control character in the line that you are editing, precede this control character by **CONTROL-V** (Verbatim). You can use this **CONTROL-V** sequence either to replace or insert control characters in the line. If you type any nonediting control character without first preceding it with **CONTROL-V**, your Apple II beeps.

Control characters are always displayed in inverse video so they stand out on the screen. You should not use control characters in your files, since the Assembler does not accept control characters in assembly-language source files.

You can type lowercase characters if you enable this feature before you enter Edit mode. If your particular system requires them, **CONTROL-E** and **CONTROL-W** work as the normal Enable/Disable Lowercase commands. If you have lowercase characters in your text, these characters will appear as blinking inverse-video uppercase characters when you place the cursor over them in Edit mode. This does not mean that these characters are changed in your text buffer; they will appear normally as soon as you move the cursor to another character.

When you position the cursor over a tab character (normally **SPACE**), the cursor jumps over the empty space caused by the tab character. If you are inserting characters before a tab character, this empty space gradually fills in as you type more characters. If you replace the tab character with another character, the line on the screen may appear to jump to the left as the tab's empty space is eliminated.

In Edit mode, **CONTROL-F** finds a specified character on the line you are editing.

You can jump quickly within the line you are editing when you use the Edit mode character-search command: **CONTROL-F** followed by a character moves the cursor to the first occurrence of that character to the right of the original cursor position. If the indicated character does not occur on the edit line, the cursor remains in its original location.

**CONTROL-R** restores an altered line to its original form.

You can restore altered characters on the edit line. Press **CONTROL-R** at any time to restore the line to its original form; this lets you re-edit the line from scratch.



When you are satisfied with an edited line as displayed on the screen, press **(RETURN)**. This places that line back in the text buffer and presents the next line for editing. When you indicate, by pressing **(RETURN)**, that you are through editing all the lines that you specified in the Edit command, you return to the Editor command level.

Use **(CONTROL)-(T)** to save only the first part of an edited line.

If you inserted some text and want to save only the first part of the line that you edited, place the cursor to the right of the text that you want to save, and press **(CONTROL)-(T)** to truncate the line. Only the text to the left of the cursor is placed in the text buffer. The next line then appears for editing.

Use **(CONTROL)-(X)** at any time to cancel Edit mode and return to the Editor command level.

If you want to cancel Edit mode at any time and return immediately to the Editor command level, press **(CONTROL)-(X)**. This leaves the current line in its original form in the text buffer and returns you to the command level. If you changed the line displayed on the screen, that line in the text buffer remains unchanged but the line displayed on the screen is not restored to its original form.

Table 2-1 gives a summary of the Edit mode keystroke commands.

**Table 2-1.** Edit Mode Keystroke Summary

Editing Function	Edit Mode Keystroke
Move cursor <b>left</b> one character	<b>(←)</b>
Move cursor <b>right</b> one character	<b>(→)</b>
<b>Delete</b> current character	<b>(CONTROL)-(D)</b>
<b>Insert</b> characters at this position	<b>(CONTROL)-(I)</b>
<b>Replace</b> a character	any noncontrol character
Enter control character into text ( <b>Verbatim</b> )	<b>(CONTROL)-(V)</b> any character
<b>Restore</b> the original line	<b>(CONTROL)-(R)</b>
<b>Find</b> the indicated character in the line and move the cursor there	<b>(CONTROL)-(F)</b> any character
<b>Store</b> line as it appears on screen	<b>(RETURN)</b>
<b>Truncate</b> at current cursor and store text in text buffer	<b>(CONTROL)-(T)</b>
<b>Cancel</b> Edit mode and return to command level	<b>(CONTROL)-(X)</b>



The **Swap** command allows you to edit two files (in separate text buffers) at once, letting you jump back and forth between the two buffers.

---

### **Editing Two Files at Once**

Often when you are editing assembly-language source files, you may want to check the contents of another source file, or to make changes to another file, without saving and reloading your current editing file. The Editor lets you edit two files at once by splitting the text buffer into two parts. If there is sufficient room in the two resulting buffers, you can then edit the two text files concurrently, alternating between the two buffers by using the Swap command.

This feature is useful if you are working on an assembly program that spans multiple files. Very often, you want to reference quickly other files that are also in the source program to make corrections and to incorporate changes.

If you are currently editing one text file and want to edit a second one concurrently, type

SWAP

and press **(RETURN)**. This *splits* the current text buffer and stores the current filename, slot, drive, volume, and file size in memory. The original text buffer is split at the end of the current text and becomes text buffer #1. Text buffer #2, which is empty, becomes the current text buffer.

You can now load, edit, list, save, or perform any other editing operations on the contents of text buffer #2, while text buffer #1 remains in memory. If you want to go back to the text in text buffer #1, just type

SWAP

and press **(RETURN)** a second time. The Editor then makes text buffer #1 the current text buffer, and preserves the contents of text buffer #2.

While you have two text buffers in memory, you can perform any editing function. You can identify which text buffer you are currently editing by the buffer number, either 1 or 2, that appears before the command prompt on every command line.

The ASM command is discussed in Chapter 3.

You cannot call the Assembler using the ASM command while you have a second text buffer active. If you try to do so, the screen displays **MULTI BUFFER ERROR**. You cannot call the Assembler until you deactivate text buffer #2.

The **KILL2** command deactivates text buffer #2, and clears any text that was in this buffer.

To deactivate text buffer #2 and delete any information that is in it, type

**KILL2**

and press **(RETURN)**. If you wish to save the contents of this buffer, you should do so by typing the Save command while you are currently editing text buffer #2. You can type the Kill2 command while you are editing either text buffer.

As an alternative, if you wish to transfer the contents of text buffer #2 into text buffer #1 and then deactivate split-buffer mode, you can do so by typing the New command while you are currently editing text buffer #1. This command clears the contents of text buffer #1. Then to activate the contents of text buffer #2, type the Swap command. If text buffer #1 is empty when you type this Swap command, split-buffer mode becomes inactive and the old contents of text buffer #2 become the new contents of the single remaining text buffer.

---

### ***Altering the Display***

The commands discussed in this section control various display features of the Editor. None of these commands alter the contents of the text buffer in any way.

#### ***Setting Tabs***

When you first run the Editor program, the standard tab positions for formatting the appearance of 6502 assembly-language source files are in effect. Initially, the space character is used as the tab character, and the tab columns are set at 16, 22, and 36. If you are using the Editor to create other types of text files, you probably want to change these tab settings and the tab character.

To change the tab settings, type

```
Tabs [tabcol [, tabcol [, tabcol [...]]]]  
[ , <tabchar> . ]
```

and press **(RETURN)**. You can specify up to ten tab positions following the Tabs command, separating each tab position with a comma. You must type these positions in increasing order for them to function properly. If you type a delimiter (shown as a period above) followed by a different single character and the delimiter, this single character becomes the new tab character. If you type the Tabs command without any parameters, you turn off the tabbing function.

The **Tabs** command lets you change the tab settings in effect when you begin using the Editor program.

Use only **(SPACE)** or **(CONTROL)-I** as the tab character when you are editing assembly-language source files.

You can change the tab character to any character that you like. If you are editing assembly-language source files, you *must* use either the space character or **(CONTROL)-I** as the tab character. If you are using an Apple IIe system, **(TAB)** functions the same as pressing **(CONTROL)-I**. The Assembler recognizes only these two characters as valid tab characters.

The Editor recognizes tab characters only in the first 40 columns of each text line. If you set any tab positions beyond column 39, these tab positions are ignored by the Editor, although the Assembler will still use these tabs to arrange its output listings.

The Editor uses the tab positions when text lines are displayed in response to a List, Print, or Edit command. Tabs are not active when you are in Input mode: when you type a space, it appears as a single space. When you later list this line, the space is treated as a tab character and may be displayed as several blanks on the screen.

Tabs also cause a slightly different appearance for lines when you print rather than list them, since a six-column relative line number is not displayed when you print lines. Under most conditions, the same line appears with six extra spaces between the first two fields when you use the Print rather than the List command.

#### ***Using a 40- or 80-Column Display***

If you have an Apple IIe with the Apple IIe 80-Column Text Card, or an Apple II or Apple II Plus with an ALS Smarterm 80-Column Card in slot 3, then the Editor automatically comes up in 80-column uppercase and lowercase display mode. If you wish to alternate between 80-column and 40-column display modes, you type

COLumn 40

and press **(RETURN)** to enter 40-column mode, and you type

COLumn 80

and press **(RETURN)** to return to 80-column mode. If your Apple II system does not have 80-column display capability, these commands are ignored.

Use the **Truncate-on command** to see only the first 40 characters of each text line when listing or printing lines.

### **Truncating the Display**

If you are using a 40-column display to create assembly-language source files, you can selectively list or print only the first three fields of your source statements, truncating the comment field so that each statement can be displayed only on a 40-character line. When you type

```
TRuncON
```

and press **(RETURN)**, from that point onward you see only the characters in your source statements up to the first space-semicolon sequence in the line. This two-character sequence is how all 6502 comments begin when you include them at the end of your source lines. Since these comments typically wrap to a second line when you use 40-column display mode, you can make your source programs much easier to read by using the Truncate-on command. This command in no way affects the actual text in the text buffer; it only affects what you see on the screen when you type a List or Print command. This feature does not affect the operation of the Find, Change, or Edit commands.

To restore the Editor to its normal display mode, type

```
TRuncOFF
```

and press **(RETURN)**. This turns off the line-truncation feature that you earlier activated with the Truncate-on command.

Truncation-on mode is automatically suspended when you are using Edit mode so you will not accidentally lose your comments because they are not displayed.

The **Truncate-off command** deactivates an earlier Truncate-on command.

The **End command** lets you leave the Editor and return to Applesoft BASIC.

### **Leaving the Editor and Returning to BASIC**

When you finish editing a program and have saved your work, you can return to the BASIC command level (immediate mode) by typing

```
END
```

and pressing **(RETURN)**. When you type the End command, the Editor simply returns to BASIC; none of the text that you have been editing is saved or altered.



#### **Warning**

The Editor does not automatically save your work; you must use the Save command before you leave the Editor if you want to preserve the program you have been editing.

If you type the End command by accident, you can usually recover the work you were editing. Immediately type after the BASIC prompt

MAXFILES 5

and press **(RETURN)**. Then type

CALL 3075

and press **(RETURN)**. You are now at the Editor command level.



#### Warning

When you return to the Editor, examine your edit file completely. Do not save this edited file until you are sure that it is intact, or you might replace the intact old version of your file on a disk with a scrambled new version.

The **Monitor-on** command lets you leave the Editor and enter the Apple II Monitor. Typing 3000 or **(CONTROL)-(Y)** returns you to the Editor.

Memory areas used by the Editor/Assembler: see Appendix H.

#### Entering the Monitor

You can leave the Editor and enter the Monitor by typing

MON

and pressing **(RETURN)**. Normally you do not use this command unless you want to use one of the Monitor commands. To return to the Editor command level from the Monitor, type 3000 or **(CONTROL)-(Y)**. If you have an 80-column display card installed, use 3D0G.

When you are in the Monitor, you can use any of the Monitor commands. If you expect to return to the Editor, you must not disturb any of the memory areas used by the Editor/Assembler system.



#### Warning

Note that if you enter either BASIC from the Monitor, you will destroy memory areas used by the Editor/Assembler. The Monitor-on command is a tool for the experienced Apple programmer and would be better left unused by the beginner.

#### Identifying the Absolute Location of Text in Memory

To manipulate text from the text buffer using the Apple II Monitor, you may sometimes find it useful to know the absolute location of a particular text line in memory. To do this from the Editor command level, type

Where line#

and press **(RETURN)**. You can use this command to display the absolute memory address of the first character of the indicated line of text. It also provides a means of discovering the current memory address of the beginning of the text buffer; to find this, type

Where 1

and press **(RETURN)**. Normally, you see this display

1=\$2800

which is the same as 10240 decimal, the normal starting location of the Editor's text buffer.

# ***The 6502 Assembler***

---

<b>51</b>	<b>Introduction to the Assembler</b>
<b>52</b>	<b>A Tutorial: Using the Assembler</b>
<b>52</b>	Getting Started
<b>53</b>	Assembling Your Program
<b>55</b>	<b>Using the Assembler</b>
<b>55</b>	Calling the Assembler
<b>56</b>	Suppressing the Generation of Your Object File
<b>57</b>	Coresident Assembly With a 64K Apple II
<b>57</b>	Recovering From Errors
<b>58</b>	Stopping Your Assembly
<b>59</b>	The ASMIDSTAMP File
<b>59</b>	Generating Assembly Listings
<b>59</b>	Selecting a Printer to Receive Your Assembly Listings
<b>61</b>	Interpreting Your Assembly Listing
<b>63</b>	Listing Tab Control
<b>63</b>	Interrupting the Listing
<b>63</b>	Turning Off the Assembly Listing
<b>64</b>	The Symbol Table Listing
<b>65</b>	<b>Assembly-Language Source Files</b>
<b>65</b>	The Syntax of Assembly Statements
<b>66</b>	The Label Field
<b>67</b>	The Mnemonic Field
<b>68</b>	The Operand Field
<b>68</b>	Registers
<b>68</b>	Identifiers
<b>69</b>	Constants
<b>70</b>	Expressions
<b>70</b>	High-Byte and Low-Byte Operators
<b>71</b>	The Location Counter
<b>72</b>	Zero-Page Addressing
<b>72</b>	The Comment Field



73	Giving Directions to the Assembler
74	Controlling the Overall Assembly
74	ORG
75	DSECT
76	DEND
77	OBJ
77	REL
78	SW16
78	Assigning Information
79	EQU
79	DEF or ENTRY
80	ZDEF
80	EXTRN or REF
81	ZXTRN or ZREF
81	A Note on External Symbols
81	Generating Data in Your Object Code
82	DFB or DB
82	DW
82	DDB
83	DS
83	MSB
84	ASC
84	STR
84	DCI
85	DATE
85	IDNUM
85	Controlling Conditional Assembly
85	DO, IFXX, ELSE, and FIN
87	FAIL
88	Source File Directives
88	CHN
89	INCLUDE
89	SBUFSIZ and IBUFSIZ
90	MACLIB
91	Controlling Your Assembly Listings
91	PAGE
92	LST
93	Cycle Times
93	Generated Object Code
93	Warnings
94	Unassembled Source
94	Expanded Macro Source
94	Alphabetic Symbol Table
94	Value-Ordered Symbol Table
95	Sixup Symbol Dump

95	REP
95	CHR
96	SKP
96	SBTL
96	Using Macros in Your Assembly-Language Programs
97	Calling Macros in Your Program Source File
97	The Macro Definition File
99	The &0 Parameter
100	The &X Parameter

## ***The 6502 Assembler***

The 6502 Assembler in the Tool Kit Volume II is a powerful tool that lets you program your Apple II system in assembly language. You use the Assembler to translate the assembly-language source files that you create using the Editor into executable object programs that will run on your Apple II. The Assembler can also help you locate problems that may occur in your programs by generating assembly listings, error summaries, and symbol table listings to accompany your programs.

This chapter tells how to use the Assembler, but does not describe how to program in 6502 assembly-language. Several excellent reference manuals on 6502 programming are listed in the Preface. You should be familiar with one of these books before you continue reading this chapter.

- The first part of this chapter describes some of the characteristics of the Assembler.
- A brief tutorial on using the Assembler follows. It leads you through an assembly of the program source file that you created as part of the Editor tutorial in Chapter 2.
- The remainder of this chapter introduces the structure of the assembly-language source files that you use as input to the Assembler program. The syntax of assembly-language source statements and the various Assembler directives that you can use are all described in detail.

Appendix B contains a summary of the 6502 assembly-language mnemonics, address-mode syntax, and assembly directives to use in assembly-language programs.

### ***Introduction to the Assembler***

The Assembler creates an executable object program by translating each of the assembly-language statements in your program source file into an executable machine operation code **opcode**. These assembly-language statements can include any of the standard 6502 assembly-language mnemonics and addressing syntax.

In addition, you can include an extensive set of Assembler **directives**, or instructions to the Assembler itself, in your program source file. The Assembler has a limited macro capability that you can use to make your programming more efficient. You can also chain several source files together to assemble modular or very large 6502 programs. All of these features are described in this chapter.

The Assembler can create both binary object programs and relocatable object programs. Binary programs are programs that you can run directly using the DOS BRUN command. Relocatable programs are assembly-language programs that you can load and run during the execution of a BASIC program. Relocatable object programs are discussed later in this chapter and in the chapter on the Relocating Loader.

### ***A Tutorial: Using the Assembler***

This brief tutorial shows you how to assemble your assembly-language source file using the Assembler. When you finish with this tutorial, you will have

- assembled the assembly-language source file you created in the tutorial in Chapter 2;
- stored the resulting binary program on a disk.

To work through this tutorial, you need:

- your Apple II computer running the EDASM program
- the TESTPROGRAM file that you created during the Editor tutorial in Chapter 2.

You will use the Assembler to assemble the TESTPROGRAM source file you created using the Editor. The Assembler saves the assembled object program as a binary file on your disk using the name TESTPROGRAM.OBJ0.

You will use this executable binary program when you work the Bugbyter tutorial in Chapter 4. For this reason, follow the steps in this tutorial, if only to prepare the necessary file you will use in the next tutorial.

---

### ***Getting Started***

The Assembler is an integral part of the combined Editor/Assembler program; you can call the Assembler only from the Editor command level. If your Apple II is not currently running

See the section "Getting Started" in the tutorial in Chapter 2 for a description of how to run the Editor/Assembler and how to use the Editor command level.

the Editor/Assembler, follow the Editor tutorial to start up the Editor/Assembler and place yourself in the Editor command level. When you see the colon (:) prompt character at the left margin of your screen, you are at the Editor command level.

Make sure your assembly-language source program is on the DOS Programmer's Tool Kit Volume II disk.

What You Do	What Happens
1. Type the Editor command  CAT  and press <b>RETURN</b> .	The Editor displays the contents of the current disk, which should contain your program as a text file named TESTPROGRAM.

**By the Way:** If your program is not shown in the disk contents, check that you are reading the contents of the DOS Programmer's Tool Kit Volume II disk. If your program is not on this disk, then perhaps you did not complete the Editor tutorial in Chapter 2. Return there now and complete the steps to create and store your assembly-language program on the disk. When you finish, return to this tutorial to assemble your program.

## Assembling Your Program



### Warning

Since the Assembler overwrites the Editor's text buffer as it assembles your program, you must save any edited text and clear the text buffer before you use the Assembler. (For an exception to this rule, see the section "Coresident Assembly With a 64K Apple II" in this chapter.) The Save and New commands to do this are described in Chapter 2. If you do not clear the text buffer before calling the Assembler, you see a BUFFER ERROR message.

When you have verified that your source program is on the proper disk, you can use the Assembler to assemble your program.

1. From the Editor command level (the colon prompt), type  ASM TESTPROGRAM  and press <b>RETURN</b> .	The Assembler reads your program source file TESTPROGRAM from the current disk and assembles it to produce a binary file. The Assembler saves this binary file to the disk using the name TESTPROGRAM.OBJ0.
---	---

If you are using an Apple II system without an 80-column text card, your screen shows only the first 40 columns of this listing.

The Assembler also generates a complete assembly listing of your program on your video screen. This assembly listing looks like this:

```

SOURCE      FILE: TESTPROGRAM
--- NEXT OBJECT FILE NAME IS TESTPROGRAM.OBJ0
1000:      1000      1      ORG      $1000
1000:A0 C0      2      START      LDY      #$C0
1002:A2 00      3      LDX      #0
1004:20 0D 10      4      LOOP      JSR      STORE
1007:E8      5      INX
1008:E0 05      6      CPX      #5
100A:D0 F8      7      BNE      LOOP
100C:60      8      RTS
100D:C8      9      STORE      INY
100E:98      10      TYA
100F:9D 00 11      11      STA      BUFF,X
1012:60      12      RTS
1013:      00ED      13      DS      $ED,$00
1100:      000A      14      BUFF      DS      10,$00
      1100 BUFF      1004 LOOP      01000 START
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 18-SEP-82 000020
** TOTAL LINES ASSEMBLED 15
** FREE SPACE PAGE COUNT 79

```

**Didn't Work?** If you made any typing mistakes when you typed the ASM command, the Editor/Assembler displays an error message and returns you to the Editor command level. Just retype the ASM command correctly and try again.

If the Assembler cannot find your TESTPROGRAM source file, use the Editor's Catalog command to verify that your file is on the disk in the currently-selected drive.

If the Assembler found any errors in your TESTPROGRAM source file during its assembly, you probably made a typing mistake when you created this file. Go back to the Editor tutorial now and make sure that your file matches the text shown in the tutorial. If not, use the Editor to fix any mistakes before you try to assemble your program again.

See Appendix D for explanations of the error messages.

2. To verify that your binary object file is stored on the disk, type

CAT

and press **(RETURN)**.

The Editor shows the contents of your disk, with TESTPROGRAM.OBJ0 shown at the end of the other files on the disk.

```

DISK VOLUME 100
A 004 HELLO
B 012
B
.
.
.
T 002 TESTPROGRAM
B 003 TESTPROGRAM.OBJ0
.

```

You have now finished the Assembler tutorial. You learned how to use the Assembler

- to assemble your assembly-language source file and produce an executable binary object program;
- to store the binary program onto a disk;
- to generate an assembly listing of your program.

The TESTPROGRAM.OBJ0 binary (B) file that you produced in this tutorial will be used in Chapter 4 as you learn how to test and debug your assembly-language programs.

## Using the Assembler

The Assembler tutorial demonstrates a very simple use of the Assembler: to assemble your program source files from a disk and to store the resulting object program on a disk. This is probably the most common use of the Assembler. You can also use the Assembler in several other ways. In the sections that follow, each of these additional techniques is described in detail.

---

### Calling the Assembler

To call the Assembler from the Editor command level, type the ASM command using the following syntax:

```
ASM srcfile [.objfile] [.[SSS] [.[DDD]d]
[.WWW] ]]
```

- The source filename (designated srcfile above) must be the name of a valid assembly-language source file in the Editor's current slot, drive, and volume.
- You may also specify an object filename (designated objfile above), and an object file slot, drive, or volume. You need to specify this information only if you want the object file to be stored under a different name or on a different disk.
- If you do not specify an object filename, the Assembler creates an object filename by appending the characters .OBJ0 to the source filename. If you do not specify an object file slot, drive, or volume, the Assembler stores the object file on the same disk from which it read your program source file.
- If you specify a Volume parameter, the Assembler directs the object file to the disk bearing this volume number. If this volume is not present in a drive when the Assembler attempts to open the object file, the Assembler prompts you to insert this disk. If

The current slot, drive and volume settings of the Editor are discussed in Chapter 2 in the section "The Current Disk."



you specified an incorrect volume number and the Assembler prompts you to insert an incorrect disk, press **(CONTROL)-(C)** to end the prompting and allow you to start again.

Remember, the optional parameters are positional and must occur in the order shown. If you want to use the default values of some parameters but specify one of the later ones, type a comma in place of each defaulted parameter. For example, if you want to specify only the drive of the object file and use the default object name and slot, then type this ASM command

```
ASM PROGRAM, , 02
```

Note that you *must* include commas in the ASM command to indicate the omitted object filename and disk slot parameters. The labels **S**, **D**, and **V** are optional—the numbers themselves will suffice—but the labels make it easier to remember the purpose of each parameter.

**By the Way:** If you have a 48K Apple II system, before you call the Assembler, make sure that the Editor/Assembler system disk is still in drive 1. The Editor has to load the Assembler program from the disk before it can assemble your program.

If you have an Apple II system with 64K of memory or more, the Assembler program is always in memory when the Editor/Assembler program is running. It does not have to be loaded from the disk.

### ***Suppressing the Generation of Your Object File***

When you are assembling your program simply to get a listing or to check for errors, you can speed up the assembly process considerably by suppressing the generation of your object file on the disk. To select this assembly option, you must specify an *at* sign (@) in place of the object filename when you type the ASM command. For example

```
ASM TESTPROGRAM, @
```

This command assembles the TESTPROGRAM file found in the current disk slot and drive, and generates assembly and symbol table listings, but does not produce an object file on the disk.

You can have the Assembler store your object code directly into memory by using *at* in the ASM command in conjunction with an OBJ directive in your source program. This little-used feature of the Assembler is discussed in the description of the OBJ directive later in this chapter.

**Coresident assembly:** both your source text and the Assembler reside in memory at the same time.

### ***Coresident Assembly With a 64K Apple II***

If you are using an Apple II system with at least 64K of memory, you can speed up the assembly of your programs by having the Assembler take your source text directly from the Editor's text buffer. Since the Assembler does not have to read your source program from a disk, this type of assembly is very quick.

This option is called **coresident assembly**, because both the Assembler and your source text reside in memory at the same time. You cannot use any CHN (chaining), INCLUDE, or MACLIB directives in your source text if you wish to use coresident assembly. Neither can you use coresident assembly with a 48K Apple II system, since there is not enough memory to store both the Assembler program and a full Editor text buffer.



### **Warning**

You should be very careful to save your source program text to a disk before using the Assembler, because the Assembler overwrites the text buffer after it has assembled your program.

To specify coresident assembly when you type the ASM command, type an asterisk (\*) in place of the assembly-language source filename. The Assembler then takes the current contents of the Editor's text buffer as the source file for this assembly. You must specify the name of the object file that is to be generated, or you can suppress the generation of an object file as described above.

For example, to initiate a coresident assembly, type

```
ASM *, TEST.OBJ0, .02
```

This command assembles the contents of the text buffer and stores the resulting object file on the disk in drive 2 of the current disk slot. If you do not specify a filename for the object file when using coresident assembly, the Assembler stores the object file on the current drive using the name OBJ0.

---

### ***Recovering From Errors***

If you make any syntax errors when you type the ASM command, the Editor/Assembler shows you an error message and places you at the Editor command level.

Appendix D contains an explanation of the DOS errors that may occur during operation of the Assembler.

If the Assembler cannot locate a source file you specify, or some DOS error occurs, the Assembler aborts its assembly and returns you to the Editor command level. Before returning, the Assembler displays the message

```
ASSEMBLY ABORTED . PRESS RETURN
```

and waits for you to respond by pressing any key. To make sure that the Assembler has not aborted without closing all the open files, it is a good idea to use the .CLOSE command to make sure that any open files are closed.

During assembly of your program source file, the Assembler displays individual messages for any errors it finds. These error messages appear on your screen or on your printer, depending on where you directed your assembly listings.

---

### Stopping Your Assembly

You can stop the Assembler at any time during an assembly by pressing (CONTROL)-(C). When you press (CONTROL)-(C), the Assembler closes all open files and frees any DOS buffers that are in use before returning you to the Editor command level. The Assembler does not remove any of the output files that it may have generated on your disk. To remove these files, use the direct .DELETE command from the Editor command level.

There may be occasions where the Assembler will "hang up" after you type (CONTROL)-(C) to stop an assembly. For example, this can occur if the Assembler is directing output to a printer that is not on-line. To recover press and release (RESET) twice, pausing between keystrokes. The Assembler terminates *everything* and returns you to the Editor command level.

---

### Warning

Pressing (RESET) during an assembly while the object file is being stored on a disk can be very, very dangerous for your disks. If the Assembler is in the middle of allocating output file space in the Disk Vtoc at the time you press (RESET), you can destroy access to your entire disk. *Do not press (RESET) during an assembly while the light on any disk drive is on; that is, while data is being written to a disk.*

---

---

### **The ASMIDSTAMP File**

You may recall from Chapter 2 that the Editor/Assembler maintains an ASMIDSTAMP file to keep track of your source disks. When you type the ASM command, the Assembler loads this file from the Editor's current disk slot, drive, and volume.

If an ASMIDSTAMP file doesn't exist on this disk, the Assembler displays a warning `FILE NOT FOUND` message, although the Assembler continues assembling your source file.

If there is an ASMIDSTAMP file on the disk, the Assembler increments the last six digits of this file and then saves the ASMIDSTAMP file back onto your source disk. If this disk is write-protected, the Assembler displays a `WRITE PROTECTED` message and continues assembling your program.

The Assembler uses the data from the ASMIDSTAMP file when you use the `DATE` and `IDNUM` directives in your program source file. These directives are described in the section "Controlling Your Assembly Listings." They allow you to mark your object files and assembly listings so that you can later associate each of your printed assembly listings with a particular object file, even if you assembled the same source program several times during a single programming session.

You can copy the ASMIDSTAMP file from the DOS Programmer's Tool Kit Volume II disk onto your own program source disks using the FID program on your DOS System Master disk.

---

### **Generating Assembly Listings**

In the tutorial earlier in this chapter, you viewed your assembly listing on your Apple II video screen. The following sections describe how you can direct these listings to a printer, and also explain how you can control which parts of your assembly are printed in your listings.

#### **Selecting a Printer to Receive Your Assembly Listings**

Before you call the Assembler from the Editor command level, you can select where the Assembler is to direct your assembly listings; you can direct these listings to a printer or other device if you prefer. If you don't select a particular device to receive your assembly listings, the Assembler displays them on the Apple II video screen.

See the *DOS User's Manual* for a description of how to copy a file.

The **PR#** command lets you select a printer to be used later for your assembly listings. Note that the Editor's **PR#** command is different from that of Applesoft/DOS. Do *not* type a period before this **PR#** command.

To select a printer or other device to receive your assembly listings, you must type the **PR#** command from the Editor command level. This command has the following syntax:

**PR# slot# [ , [L#] [P#] [device-control-string] ]**

After you type this command, any assembly listings generated during the current Editor/Assembler session are directed to the specified device. Normally, you send your listings to a printer connected to a printer interface card, but you can also direct them to an 80-column terminal or some other output device.

The slot number (slot#) you type specifies the interface card slot controlling the output device that will receive your listings. If you specify a slot number of zero, the Assembler directs your listings to the Apple II video screen.

---

#### Warning

If you specify **PR# 6** (when your disk controller card is in slot 6) and later attempt to call the Assembler, the Editor/Assembler will stop and you will have to reboot. Always be careful to select the appropriate device when specifying an Apple II slot number.

---

The Assembler can direct your assembly listings to only one output device at a time. It is possible that the output device you select may echo what it is receiving back to the normal Apple II 40-column video-output routine, but this is up to your device and how you use it. The Apple II printer interface cards cannot print a listing that is wider than 40 columns and simultaneously echo to a 40-column Apple II screen.

There are three other optional parameters that you can specify with the **PR#** command:

- The first optional parameter is a **logical-page length** that indicates the number of lines you want the Assembler to print on a page. Unless you specify differently, the Assembler prints 60 lines. You can specify a different logical-page length by typing **[L]** followed by a two-digit number.

- The second optional parameter is a **physical-page length**, that is, the number of lines from one top-of-form to the next. You can specify the physical-page length by typing (P) followed by a two-digit number. If you used a SBTB directive in your source file and you do not specify a physical-page length, the Assembler outputs a form-feed character after each page.
- The third optional parameter is a **device-control-string**, which is any sequence of control characters that your printing device requires to initialize it before printing the Assembler's output listing. This device-control-string may not exceed 32 characters in length. For some Apple II printer interface cards, this typically consists of the normal (CONTROL)-(I), (N) sequence that turns off the Apple II screen and initializes the printer. You can type this sequence by holding down (CONTROL) while you type (I), then releasing both keys and typing (N).

When you type the PR# command, the Editor/Assembler preserves this information in memory and uses it for the remainder of your current session. When you begin a new session with the Editor/Assembler, you must again specify where to direct your listings. A most convenient way to call the Assembler is to include this command with the Editor ASM command in a DOS EXEC file. You can then call the Assembler by executing the DOS EXEC command from the Editor command level.

An example of the PR# command, as you would type it at the keyboard, is

```
PR#1, L54 P66 *N
```

where \* represents (CONTROL)-(I). This command tells the Assembler to direct any assembly listings generated during this session to the printer with an interface card in slot 1. Listings will be printed with 54 lines on a page on paper that is 66 lines in length.

### ***Interpreting Your Assembly Listing***

You can direct the Assembler's output listing to either the Apple II video screen or to an output device having an interface card in an Apple II slot. When you direct the listing to a device other than the video screen, the Assembler assumes that at least 80 columns of



print line-width are available for the listing. If your source lines and tabs are set so as to print beyond column 80, your printer must either wrap the lines itself or be able to receive more than 80 columns of output from the Assembler. If you use at least one SBTLL directive, headers are printed at the top of each listing page. The header consists of a title line followed by a blank line, as shown in Figure 3-1.

Figure 3-1. A Typical Assembly Listing

```
-----
01 CMDINTPR      COMMAND INTERPRETER      18-JUN-81 #000010 PAGE 30

---- NEXT OBJECT FILE NAME IS OBJ
0000:      0000 79      OPC $0000
0000:      0000 80 EDITOR  EQU $0000
0000:      0000 81 ASSEM   EQU $0000      Bank 2
0000:      82 *****
0000:40 15 00      84 COLD    JMP TEXT      Hard startup
0000:40 78 00      85      JMP CTXT      SOFT START

0015:43 FF      94 TEXT      LDI #FF
0017:9A      95      TNS      Force stack
0018:20 58 FC      96      JSR HOME
0018:A0 00      97      LDI #0
0010:B9 08 12      98 SENDBANK LDA BANNER Y
0020:F0 0E 0028      99      BEQ BANKEND
0022:20 ED FD      100     JSR COUT
0025:C8      101      INY
0026:D0 F5 0010      102     SNE SENDBANK

1308:30 24 30      371 MINTAB OFS 22 76.48 minimum tabs
1308:      372 *****
1308:      373 *****
1308:      374 BANNER OFS $0d $0d $0d $0d $0d $0d
1311:A0 05 04 08      375     ASC      EDITOR ASSEMBLER II
1325:B1 B8 AD 0A      376     DATE
```

The example in Figure 3-1 illustrates many of the features of the Assembler's listing file format. The listing header contains the file number at the left, followed by the first eight characters of the file name, and the optional title. On the right are the date, ID number, and page number. Below the header is a blank line.

The Assembler prints each source statement beginning with a four-digit hexadecimal number followed by a colon. This number is the value of the Assembler's program counter (PC) when that line was assembled. Line 80 shows the expression-result field to the right of the PC field, typical of a control directive.

The Assembler shows the NEXT OBJECT . . . message when it encounters an absolute ORG directive in your source program while the Assembler is directing its object output to disk storage. Lines 84 thru 102 show how normal 6502 code is printed, with the branch target address off to the right of the branch object code.

Lines 871 thru 876 show how data directives are printed, with the NOGEN option in effect. The NOGEN option suppresses printing more than one listing line for data directives that generate more than four bytes of object output. Also note that lowercase is legal input for the Assembler; it is shifted to uppercase for internal use, but is printed as given in the source.

#### ***Listing Tab Control***

The Assembler listing is tabulated in the same way the lines in the Editor are displayed. If the Editor tab settings are reasonable values when the ASM command is executed, the Assembler uses them to format the output listing. The Assembler uses these settings if the first tab setting is 12 or more, and uses its minimum tab settings otherwise. The minimum settings are equivalent to Editor tabs of 15, 19, and 31. The Assembler uses only the first three Editor tabs. Large Editor tab settings may cause listings to be shifted off the printed page.

#### ***Interrupting the Listing***

Press **(SPACE)** to interrupt your assembly listing.

You can make the Assembler pause when it is displaying or printing an assembly listing by pressing **(SPACE)**. This interrupts the Assembler output, allowing you to view portions of the listing. To restart the listing, type any character that is not a mode character, **(CONTROL)-(O)**, or **(CONTROL)-(K)**, to the Assembler.

You can **single step** through the listing a line at a time by pressing **(SPACE)** once for each line. If you are viewing the assembly listing on a 40-column video screen, you can view the second half of the 80-column listing (columns 41-80) by pressing **(ESCAPE)**. Pressing **(ESCAPE)** a second time allows you to single step through the comment portions of your listing. To continue the assembly at normal speed, type any key other than **(ESCAPE)** or **(SPACE)**.

#### ***Turning Off the Assembly Listing***

To control the portions of your source program printed in the assembly listing, you typically use the Assembler's LST ON/OFF directive in your program source file. You can also control the listings from your Apple II keyboard during assembly. Your keyboard commands override the current state of the LST option

until the next LST directive in your source program or your next command from the keyboard. Using these keyboard commands, you can examine sections of the listing you turned off from within your source program, or you can turn off the listing of sections that your source program directed be listed:

- To turn the listing off, type `(CONTROL)-(N)`, where *N* means "not listed."
- To turn the listing on, type `(CONTROL)-(O)`.

Remember that the Assembler can encounter a LST directive within the source program that counteracts your most recent `(CONTROL)-(N)` or `(CONTROL)-(O)` command. To correct this, just retype your command from the keyboard. Normally, you use these commands only when you are working on large programs; however, when you change a small portion of your program, for example, you may not want to list the entire program.

### **The Symbol Table Listing**

After generating your assembly listing, the Assembler normally produces an optional **symbol table** listing, which can be sorted in either alphabetical or symbol-value order. You can suppress the printing of this listing if you abort the assembly by pressing

`(CONTROL)-(N)`

or if you use the LST NOASYM,NOVSYM directive in your source program. To produce *only* the symbol table listings, use the LST OFF directive at the beginning of your source file and a LST ASYM,VSYM at the end of your source program.

When the Assembler prints the symbol table, it automatically adjusts the width of the symbol table output for the Apple II 40-column screen or for a printer, which is assumed to be 80 columns. The Assembler displays the table in two columns on the screen and prints it in either four or six columns on the printer, depending on the option you select with the LST directive.

The Assembler prints each entry in the symbol table using the following format:

- The first character in each entry indicates special information about that particular symbol:

- X* indicates an external symbol (see EXTRN directive)
  - N* indicates an entry point symbol (see ENTRY directive)
  - ?* indicates a symbol defined but never referenced
  - \** indicates an undefined symbol

If an undefined symbol appears in the symbol table, your assembly must have generated one or more NO SUCH LABEL errors.

- The next four characters contain the two- or four-digit hexadecimal value defined for the symbol. A zero-page or single-byte address is identified by two spaces before the two-digit address; a two-byte address is displayed using all four digits (even if the two leading digits are zero).
- The remaining 14 characters are the first 14 characters of the symbol's name. All unique symbols appear in the symbol table, but only the first 14 characters of any symbol name are ever printed.

Figure 3-2 shows an example of a symbol table listing sorted alphabetically by symbol name.

**Figure 3-2.** A Sample Symbol Table Listing

```
00 SYMBOL TABLE SORTED BY SYMBOL      27-JUN-81  #00000 PG 02
      20 ADPTR          6F ADTBLND      20BE ALPHAB      24 CH
      02000 SYNDUMP      0073 SYTYPE      20BE SWAP      20F4 SWIPEIT
      2E5F TABLND      12E84 TESTLBL      2 0A TLTBEG      27 WAL
      12E89 XXXXXXX      1 76 YYYAW      02000
```

## Assembly-Language Source Files

The Assembler accepts normal DOS text files as its assembly-language source input. A normal DOS text file is a file of logical records, where each logical record is a sequence of ASCII characters terminated by an ASCII carriage return (\$8D). Each ASCII character must have its most significant bit (msb) set to a one.

There is a great deal of additional structure that your source files must contain before the Assembler is able to assemble your source file into an executable 6502 object program. Each line or logical record of your assembly-language source file must be recognized by the Assembler as a valid assembly statement. The sections that follow discuss the syntax of these assembly statements.

## The Syntax of Assembly Statements

Each line of an assembly language source file is called an *assembly statement*. Assembly statements consist of either an *instruction statement* or an *Assembler directive*. The Assembler allows only one assembly statement per source line. Assembler directives and instruction statements follow the same general syntax.

Any text file that you create using the Editor is accepted by the Assembler as a valid input file.

An assembly statement is divided into four fields:

- the label field
- the mnemonic or operation field
- the operand field
- the comment field.

These fields are arranged in the following order

[label] mnemonic [operand] [comment]

An example of an assembly statement is

```
BEGIN LDM #64 : load size of buffer to be cleared
```

Notice that all the fields are optional except the mnemonic or operation field. The only assembly statement that cannot contain anything in the mnemonic field is a pure comment line. This special assembly statement is a line where the first character is an asterisk or a semicolon.

Although the Editor allows you to use any character as an Editor tab character, the Assembler recognizes only the space or tab ((CONTROL)-(I)) character to separate fields.

These fields must be separated by a space or tab ((CONTROL)-(I)) character. To skip over a field, use two spaces or tab characters in the assembly statement. When you use the Editor to create your program source file, the Editor uses these tab indicators to format the display of your program.

### **The Label Field**

The label field is optional in an assembly statement. Any assembly statement except a pure comment can have a label. To label an assembly statement, you must place an **identifier** starting in the first character position of the line.

An **identifier** is a symbolic name or character string that represents a 16-bit numeric value.

An identifier is a symbolic name that represents a 16-bit numeric value. Whenever you place an identifier in the label field of an assembly statement, you are defining a numeric value to be associated with that identifier. This numeric value is either the memory address that is the current value of the Assembler's program counter, or the value of the operand expression if the identifier precedes an EQU directive. The Assembler issues a DUPLICATE SYMBOL error if you attempt to define the same identifier more than once in a program.

An identifier is a character string starting with a letter and containing only letters, digits, and periods. All the characters in an identifier are meaningful, except that lowercase and uppercase letters are treated the same for all purposes except printing.

There is no arbitrary limit to the length of an identifier you can use with the Assembler. Longer identifiers tend to slow down the assembly of your programs, since it takes time to compare all the characters. They also require more space in the symbol table. As a practical matter, you should limit identifiers to eight or ten characters.

The one-character identifiers *A*, *X*, and *Y* represent registers in the 6502 microprocessor. You cannot use these identifiers to refer to any other register, address, or data in your source programs. Since some 6502 assemblers also reserve the use of the *S* and *P* identifiers, you should not use them if you want to keep your source programs compatible with other 6502 Assemblers.

You can define any absolute identifiers, or 16-bit identifiers, anywhere in your program. If you want to use a **zero page** identifier, one that refers to an address less than 256 decimal, you must define this identifier by using either a *DSECT* or *EQU* directive before you use it in an operand expression. If you use an identifier before you define it, the Assembler is forced to assume that the identifier refers to an absolute (two-byte) memory address, even if you later define the identifier to represent a zero-page or one-byte address.

It is a good practice to define all your data identifiers at the beginning of your programs. You can do this by using the Assembler's *EQU*, *DSECT*, and *DEND* directives.

#### **The Mnemonic Field**

The second field of an assembly source statement is the mnemonic or operation field, and must always contain a valid **mnemonic**. This mnemonic must always be separated from the identifier in the label field by at least one space. If you do not include a label in an assembly statement, you must include at least one space before the mnemonic.

A mnemonic consists of two or more letters terminated by a space. A mnemonic can represent one of the following:

- a 6502 assembly-language instruction
- an Assembler directive
- an assembly macro that you have defined.

A **mnemonic** is a character string that represents an instruction, an assembler directive, or an assembly macro.



A complete list of the assembly-language instructions and assembler directives recognized by the Assembler: see Appendix B.

**Assembly directives:** see the section "Giving Directions to the Assembler."

**Macros:** see the section "Using Macros in Your Programs."

The mnemonics representing 6502 assembly-language instructions are the standard MOS Technology mnemonics. The Assembler also recognizes several additional synonyms for branching instructions.

Assembler **directives**, or pseudo-operations (pseudo-ops), are instructions to the Assembler that direct the course of the assembly, define program data, reserve space, or perform other assembly chores.

An **assembly macro** is a set of assembly-language instructions that are inserted into your assembly program by the Assembler. You call a particular macro by using the name of that assembly macro in the mnemonic field of an assembly statement.

#### ***The Operand Field***

The operand field of the assembly statement is required only for some assembly instructions, Assembler directives, and macros. When you use a mnemonic for an instruction, directive, or macro that requires one or more operands, you must include these operands in the operand field, separating each operand by a comma. There must be no spaces within or between operands.

Each operand in the operand field must be one of the following:

- a register
- an identifier
- a constant
- an expression composed of constants, identifiers, and one or more operators.

#### **Registers**

Registers in an operand field refer to the 6502 microprocessor registers. You can reference these registers using the reserved identifiers A, X, and Y.

#### **Identifiers**

Identifiers are described in the section "The Label Field." When you use an identifier in the operand field, the Assembler evaluates the operand by replacing your identifier with the 8- or 16-bit value that the identifier represents. Somewhere in your program, either



before or after you use the identifier in an operand field, you must define a value for each identifier that you use. If you do not define a value for an identifier, the Assembler displays an appropriate error message.

### Constants

A constant in an assembly language program defines an explicit value. You can define two different types of constants in your programs: numeric constants or string constants.

A **numeric constant** represents a one- or two-byte numeric value. You can represent a numeric constant in any of four notations: decimal, hexadecimal, binary, or octal.

- In decimal notation, a numeric constant is represented in base ten by a positive integer between 0 and 65535, composed of digits from 0 through 9. If you use a value greater than 65535 as a numeric constant, the Assembler gives you a numeric overflow message.
- In hexadecimal notation, a numeric constant is represented in base sixteen by a dollar sign (\$) character followed by up to four hexadecimal digits. The hexadecimal digits are 0 through 9 and A through F.
- In binary notation, a numeric constant is represented in base two by a percent (%) character followed by any 16-bit binary number. This binary number is composed of the digits 0 and 1. If you use more than sixteen digits in this binary number, the Assembler gives a numeric overflow message.
- In octal notation, a numeric constant is represented in base eight by an at (@) character, followed by up to six octal digits. The octal digits are 0 through 7.

**String constants** represent sequences of ASCII characters. You can represent a string constant in your source program by enclosing the characters between single quotes: 'A' and 'AE' are valid string constants, for example. A string can be up to 240 characters in length, but all the string must be defined on a single line. When you use a string constant as the operand of an immediate-mode instruction, you need not use a trailing quote, since such an operand can be only one character.

MBS, DCI, ASC directives: see the section "Generating Data in Your Object Code" in this chapter.

When a string constant is assembled, the Assembler allocates one byte for each character. The lower seven bits of each byte correspond to the ASCII code for that character, and the most significant bit (msb) is either set or not set. You can use the MSB, DCI, and ASC directives to control the state of the most significant bit.

### Expressions

The Assembler recognizes and evaluates simple numeric expressions that you can use in place of a numeric constant or identifier. A **numeric expression** consists of constants, identifiers, and one or more operators.

The Assembler recognizes the four **arithmetic operators** +, -, \*, and / (addition, subtraction, multiplication and division) for use in expressions. When performing these arithmetic operations, the Assembler does not check for numeric overflow of the results; it just retains the 16-bit results. This lets you perform wraparound memory-address calculations using these operators.

The Assembler also recognizes three **binary-logic operators** that perform logical AND, OR, and exclusive-OR operations on two 16-bit values. The characters used to represent these operators are the caret (^), the vertical bar (|), and the exclamation point (!), respectively. Note that these are full 16-bit operators and can produce unexpected results if applied to eight-bit constants or expressions; this is especially true of the exclusive-OR operator.

### High-Byte and Low-Byte Operators

< = high-byte operator  
> = low-byte operator

The Assembler always maintains a 16-bit value when it evaluates a numeric expression. To extract an eight-bit result from this 16-bit value, you can use the high-byte (<) and low-byte (>) operators to extract the high-order or low-order byte from a 16-bit value. To help remember the meanings of the characters that represent these operators, think of them as arrowheads pointing either to the left or the right half of a hexadecimal constant. For example,

<(\$FF11) is equivalent to \$FF, and  
>(\$FF11) is equivalent to \$11.

The syntax of a valid Assembler expression, in Backus-Naur Form (BNF), is

Term := Constant , Identifier

Opr := + , - , \* , / ,  $\square$  , ! , !

Byteopr := > , <

Expression := [Byteopr] Term [Opr Term]...

This syntax definition says *An expression is a term preceded by an optional byte-operator and followed by one or more optional [operator term] sequences.* The Assembler evaluates each operator from left to right. If you do not follow this syntax when forming an expression, the Assembler signals an assembly error.

You cannot use an arithmetic or logical operator as the first element of an expression, nor can an operator be the last element. The only exceptions are the plus and minus operators (+ and -), which can be used to indicate a positive or negative expression.

Your expressions cannot contain blanks: if an expression contains a blank, the part after the blank is treated as a comment and ignored by the Assembler.

If you direct the Assembler to generate relocatable object code (using the REL directive), the Assembler will not allow multiplication, division, or the binary-logic operators to be applied to a relative identifier or subexpression. These expressions will generate a nonrelocatable result. Also, the high- and low-byte operators generally will not produce a correct relocatable result. This restriction applies *only* if you use the REL directive to generate relocatable output from the Assembler.

### The Location Counter

You can use the asterisk (\*) in an operand expression to represent the current value of the Assembler location counter. The asterisk always represents this value at the beginning of the line or statement containing the asterisk. If the asterisk follows an instruction mnemonic, the location counter points to the address of the first byte of that instruction code.

You can perform addition or subtraction operations on the value of the location counter. You can also use the high-byte and low-byte operators on this value. These operators let you align code or data structures at specific positions relative to a memory page. For example, the statements

```
HEPEL EQU *           ;define HEPEL as the low-
                        ;byte of current PC
                        DS $100-HEPEL ;fill to next page boundary
NEWPAGE EQU *          ;start of next memory page
```

create an unused data area up to the next page boundary. The code or data that follows is aligned from byte zero of the next page.

The advantage of using this method is that it always produces page-aligned code even when the size of the program preceding this code changes as revisions are made. This method can also be used to align code to other specific positions in a page and it can be used with the ORG directive to make relative ORG adjustment to the location counter.

### **Zero-Page Addressing**

The Assembler normally generates a two-byte or zero-page instruction whenever the operand is a properly defined zero-page identifier or expression. Sometimes, however, you may want to generate an absolute or three-byte instruction that refers to a zero-page address. To bypass this Assembler rule, you must equate the identifier to the desired zero-page value *only after* you have referenced the identifier at least once in an operand expression. The Assembler will then treat the identifier as an absolute identifier for the remainder of the assembly, generating absolute-addressing opcodes for all instructions that use this identifier.

### **The Comment Field**

The **comment field** is an optional field that you can use to document what your program is doing. The Assembler prints the comment field in its program listing, but for all other purposes the comment field is ignored.

A comment can contain any arbitrary set of ASCII characters and must be separated from the operand field by a space. In addition, the comment must begin with a semicolon if you want the Editor truncation facility to operate. The Assembler also recognizes statements that begin with an asterisk or a semicolon as the first character on the line. These statements are treated entirely as comments, and, although they appear in listings, they are otherwise ignored by the Assembler.

### ***Giving Directions to the Assembler***

Assembly **directives** are instructions that you put into your program source file to direct the Assembler to perform certain operations during the assembly of your program. Statements containing assembly directives resemble those containing machine instructions in that both statements can contain label, mnemonic, operand, and comment fields. Unlike machine instructions, however, assembly directives are not assembled into executable 6502 machine opcodes. Only the data-definition Assembly directives generate actual data that is included in the resulting object program.

**Assembly directives** direct the Assembler to perform certain operations during assembly of your program.

The directives that are recognized by the Assembler perform a variety of functions, including

- controlling the overall assembly of a program
- assigning information
- generating data to be included in the object file
- controlling the conditional assembly of portions of a program
- controlling any additional source files and macros that are used in an assembly
- controlling what type of object code is generated during assembly
- controlling the way assembly listings are printed.

The following sections describe each assembly directive that is recognized by the EDASM Assembler. Many of these directives are specific to the EDASM Assembler, although you will probably recognize similarities to other assemblers. The required syntax of the statement containing the directive begins each section. Where labels are not shown, you can include an optional label; any statement can also include a comment.

---

### Controlling the Overall Assembly

You use the directives described in this section to direct the overall assembly of your source program. These directives

- control the addresses at which your program is assembled;
- let you generate dummy code sections to define memory locations your program can access or constants that it can use;
- control the type of object code (load-image binary object or relocatable object code) that the Assembler generates;
- let you use and generate the Apple II SWEET16 opcode mnemonics that access the ROM-coded 16-bit pseudomachine in your Apple II system (only if you have an Apple II with an Integer BASIC ROM).

To use these directives, include the directive mnemonic like any other mnemonic in an assembly statement; some of the directives also require an operand in the statement. You can precede any directive with a label and follow any directive with a comment.

#### ORG

Form: ORG expression

The ORG directive establishes the origin address of your object code. You must use at least one ORG directive in your source file before the Assembler can begin generating object code output to a disk.

The Assembler recognizes two kinds of ORG directives: absolute and relative. A **relative ORG** is one whose operand expression contains relocatable identifiers. You must have previously defined all the identifiers that you use in the operand expression. Some examples of relative ORG directives are:

```
ORG    *+5  
ORG    SYM+10  
ORG    *-200
```

These relative ORG directives update the Assembler's internal location counter for the current object file. The Assembler updates this counter both forward and backward in relation to your object file. If you increase the value of the counter, the Assembler *fills* your object file forward to the new position with random data; if you decrease the value, the Assembler deletes any object code that it previously generated for addresses above the new value of the counter.



An **absolute ORG** directive is one whose operand contains an absolute expression, typically a constant. You use an absolute ORG directive to define the starting address the Assembler uses to generate object code. The Assembler also places this starting address in the output object file so this address can be used by DOS when you later load the object file using the DOS BLOAD or BRUN commands.

You normally use only one absolute ORG directive in your program source file. The Assembler generates a new output object file for each absolute ORG it encounters in your source file, closes the current object file, if any, and starts a new file. The filename that the Assembler uses for each new object file is generated by incrementing the last character of the previous object filename. If you want to generate multiple object files by using more than one absolute ORG in your source program, you must use these multiple object files properly, or combine them if required for later execution.

### **DSECT**

Form: DSECT

You can use the DSECT directive to define an area of memory, such as a data table or the 6502 zero page, without actually generating any object code to reside in this memory area. The DSECT directive marks the beginning of a block or group of statements in which you define the values of identifiers that you use elsewhere in your program. You use these identifiers to reference locations within the DSECT memory area. You commonly use the DSECT directive to define the labels of data items and pointers in the 6502 zero page.

The name DSECT comes from Dummy SECTION, so called because the Assembler generates no object code for this section. You can use the DSECT directive to suspend object code output temporarily and set the Assembler's location counter to address zero. Once you start a DSECT memory area, it remains in force until the Assembler encounters a DEND directive. You can include any assembly statements, including the ORG directive, within this DSECT area.

If you include an ORG directive within a DSECT, the ORG only controls the addresses that the Assembler assigns to identifiers within the DSECT. The Assembler does not treat these ORG directives as absolute ORGs, nor do they alter the Assembler's normal location counter that is saved for the duration of the DSECT area.



You cannot nest DSECT areas, that is, you cannot use a DSECT directive between the occurrences of a DSECT/DEND pair of directives. If you do, the Assembler signals a DSECT . DEND error.

### **DEND**

Form: DEND

The DEND directive marks the end of the current DSECT area. When the Assembler encounters a DEND directive, it resumes generating object code where it stopped after encountering the DSECT directive. The Assembler restores its normal location counter address that it saved during the DSECT area.

This example shows how you can use the DSECT and DEND directives:

```
                DSECT
                ORG      $000
; DEFINE ZERO-PAGE STORAGE
AREG            DS       2
BREG            DS       2
CREG            DS       2
H               DS       1
L               DS       1
                ORG      $400
TEXTPG1         DS       $400
                DEND
```

### **Warning**

If you start a DSECT area with the DSECT directive and never end this area with a DEND directive, none of the assembly statements following the DSECT directive appears in your object file output. These statements appear in listings just as in a normal assembly, but no object code is generated. This "missing DEND" problem can cause mysterious loss of object code in your output files.

## **OBJ**

Form: OBJ expression

The OBJ directive is useful only if you want to assemble your program and place the object code directly into memory rather than store it on a disk. When you suppress the generation of an object file by typing the ASM command with the @ option for the object filename, the Assembler generates no object output unless you use this directive to specify where in memory to put the object code.

The Assembler performs very specific checks on the expression given in the OBJ directive. The Assembler does not allow a value less than the current end of the Assembler's symbol table; you see the error message OBJ MEMORY CONFLICT if you attempt to use such a value. In addition, the Assembler does not allow a value greater than the HIMEM value passed to it by the Editor; in this case, or if the object code output exceeds this limit during an assembly, you see OBJ MEMORY OVERFLOW.

You cannot use the OBJ directive if you use the REL directive to generate relocatable object code output.

## **REL**

Form: REL

When you use the Relocatable (REL) directive, the Assembler generates relocatable object code. The Assembler does this by creating a **relocation dictionary** that is appended to the end of your object code. This relocation dictionary is used by the Relocating Loader program to load your object program and prepare it for execution at any starting address.

The Assembler produces a relocation dictionary only if you include the REL directive at the beginning of your source program before you use or define any identifiers.

When you use the REL directive, your object file is given a file type character of *R*, for relocatable, in the DOS disk catalog. The format of this type file is described in the Appendixes. A relocatable object file cannot be loaded by using the DOS BLOAD or BRUN commands. It can only be loaded by the RLOAD program.

Relocating Loader routines: see Chapter 5.

RLOAD program: see Chapter 5.

The REL directive requires no operand. You *cannot* use this directive in a source program containing an OBJ directive to produce coresident object code output.

When the Assembler generates relocatable object files, it clears the relocation dictionary each time it encounters an absolute ORG directive. Each segment of your assembled object code has a separate relocation dictionary.

### **SW16**

Form: SW16 [expression]

Apple II SWEET16: see Appendix G.

If you have an Apple II, this directive lets you use the SWEET16 routines that are coded in the Integer BASIC ROM. Appendix G explains how to include these routines in your own program; this allows you to use them on any type of Apple II.

When the Assembler encounters the SW16 directive in your program source file, it turns off its generation of SWEET16 OPCODE warnings and inserts a JSR instruction in your object file output. This subroutine jump is normally to the Integer Basic ROM SWEET16 entry point, \$F689. If you include the optional address expression in the operand field, the Assembler generates the JSR to that address instead.

You normally follow this directive with Apple II SWEET16 mnemonic instructions, ending with a SWEET16 RTN instruction. If you use these mnemonics in your source file without including the SW16 directive, the Assembler does not assemble them, but instead signals a SWEET16 OPCODE warning. This warning prevents your accidental use of the Sweet16 mnemonics, many of which are very similar to standard 6502 mnemonics.

SWEET16 instruction summary: see Appendix G.

---

### **Assigning Information**

You use the directives described in this section

- to assign numeric values to identifiers;
- to define the characteristics of an identifier that references or is referenced by another assembly-language program.

You include these directives—some require an operand—like any other mnemonic in an assembly statement. You can precede any directive with a label, and you can follow any with a comment.

## **EQU**

Form: identifier EQU expression

You use the Equate (EQU) directive to define the value of a symbolic identifier. The identifier that you place in the label field cannot be defined elsewhere in your program. The Assembler evaluates the expression in the operand field and defines the identifier to have this numeric value. You cannot use any external identifiers in the operand expression.

You can use symbolic identifiers

- To represent frequently used or changeable program constants, allowing you to make simple changes without having to edit many lines of the program.
- To name items of information that have special meanings. This makes your programs more readable and easier to understand and change long after they are written.

## **DEF or ENTRY**

Form: DEF identifier or ENTRY identifier

The DEF directive allows an identifier defined in a particular module of your program to be referenced as an external symbol by other modules. You can use this directive more than once in a module, either to define global identifiers or to define alternate entry points for your module.

When you use the DEF directive in an assembly statement, the identifier in the operand field is marked as a global or entry point identifier. Normally, the value of this identifier will be defined elsewhere in your program. You can, however, define the current value of the Assembler's location counter as a global entry point by including this same identifier in the optional label field of the statement containing the DEF directive. You must not use the DEF directive inside a DSECT.

Whenever you assemble a module containing the DEF directive and select relocatable object code using the REL directive, the Assembler appends an External Symbol Directory (ESD) to the Relocation Directory (RLD) of your object code output. This

external symbol directory contains all the global identifiers and external symbols defined in your module. It also includes information necessary for a Linker or Linking Loader program to link this module to other modules that may reference these entry points or global identifiers.

### **ZDEF**

Form: ZDEF identifier

You can use the ZDEF directive to create a zero-page global identifier in the same way that you use a DEF or ENTRY directive to create an absolute identifier. All the rules that apply to DEF/ENTRY apply to ZDEF except that the identifier must be a zero-page identifier. Use the ZDEF directive only after the identifier is defined as a zero-page identifier, or the Assembler will signal an **ILLEGAL LABEL** error.

### **EXTRN or REF**

Form: EXTRN identifier or REF identifier

Use the external (EXTRN) directive to indicate that an identifier is not defined in a particular module, but is instead defined externally. The Assembler always treats these identifiers, or external symbols, as two-byte identifiers, never as zero-page identifiers. If you use an external symbol as the operand of an instruction, the Assembler generates two empty bytes in the address portion of that instruction.

If you are generating relocatable object code using the REL directive and you use the EXTRN directive to define symbols as external, the Assembler adds an external symbol directory after the relocation directory in the relocatable object file.

To execute your program module containing external symbols, you must first use a Linker or Linking Loader program to link this module with one or more additional program modules where these external symbols are defined. A Linker resolves all of these external references using data from the external symbol directories of each module, and links all the modules into a single executable object program.

You can include a label identifier in the label field of the statement containing the EXTRN directive. This label is defined with the current value of the Assembler's location counter. The identifier that follows the EXTRN directive in the operand field is defined as the external symbol.

### ***ZXTRN or ZREF***

Form: ZXTRN identifier or ZREF identifier

This directive serves the same purpose for zero-page globals that EXTRN or REF does for absolute global identifiers. You must use this directive to define an identifier as a zero-page identifier before you reference it in the operand field of an assembly statement, or the Assembler will signal the error message `ILLEGAL LABEL`.

### ***A Note on External Symbols***

You can use the DEF, ENTRY, ZDEF, EXTRN, REF, ZXTRN, and ZREF directives when you are producing a relocatable object file or a binary object file. To run correctly a program that contains references to external identifiers, you must first use a Linker or Linking Loader to link this module to other program modules in which all these external references are resolved.

**Notice:** Although no such Linker or Linking Loader currently exists, these directives are provided here for completeness if you want to write your own Linker or if a Linker or Linking Loader program becomes available.

When you are creating self-modifying code, you can also make use of external symbols to define addresses that are filled in at execution time. The DEF and ZDEF directives do not cause anything to appear in the machine code portion of the object file; the EXTRN and ZXTRN directives allow undefined symbols to remain undefined without causing Assembler error messages.

---

### ***Generating Data in Your Object Code***

You use these Assembly directives to allocate or define data areas within your assembly-language program. These directives allow you to define

- byte and word data
- address tables
- data storage areas
- ASCII character data
- message strings.

You can precede each directive with a label, and you can follow each with a comment.



### **DFB or DB**

Form: DFB expr[,expr...] or DB expr[,expr...]

Use the Define Byte (DFB) directive to define one or more bytes of data to be placed in the object file. The Assembler evaluates each expression and uses the resulting value, modulo 256, as the value for each byte. If you use an identifier in the label field of the statement containing the DFB directive, this identifier represents the address of the first byte generated. If the bytes that are generated are calculated from a relocatable expression, an entry is made in the RLD for each such byte so that its value can be filled in during relocation. It is better to use multiple DFB directives—limiting each DFB directive to five to ten expressions—rather than use a large number of expressions in the operand of one DFB directive.

The comma is the only valid delimiter between expressions. You cannot use blanks between the expressions in the operand. You can use any valid expression in the operand field.

### **DW**

Form: DW expr[,expr...]

Use the Define Word (DW) directive to define two-byte 6502 words. A 6502 word is special in that the lowest eight bits of the 16-bit expression are stored in the first of the two bytes, and the most significant, or high eight bits, are stored in the second byte. You must use this order of the bytes to be able to use the 16-bit address as an indirect-address pointer in the indirect-indexed and jump-indirect 6502 instructions.

The label field identifier is given the value of the program counter, which is the address of the first, or low-order, byte of the first word. If you use more than two expressions, only the first two appear in program listings unless you turn on the LST GEN option.

### **DDB**

Form: DDB expr[,expr...]

The Define Double-Byte (DDB) directive is like the DW directive, except that the bytes are stored in reverse order, with the high-order byte first and the low-order byte second. The label-field identifier has the address of the first byte—the high-order byte—of the first double byte expression.



## **DS**

Form: DS expr[,expr]

Use the Define Storage (DS) directive to reserve a group of bytes without defining any data to be stored in these bytes. The first expression of the DS directive can contain identifiers only if you defined them earlier in your program. This expression cannot have a value greater than 16384 decimal.

If you include a second expression in the operand field, the reserved bytes are filled with the value of this expression. If you include only the first expression in the operand field, the reserved bytes contain random values.

The size of your output object module includes the amount of space reserved by the DS directive—the file is filled with data. For example, if you accidentally enter a DS with an expression that results in a value of 12K bytes, you'll suddenly get a very large output file. Use this directive only for small data areas to avoid wasting storage space on your disk. Since you do not need to store large buffers and work areas with your executable object program, do not define them by using this directive.

If you include an optional identifier in the label field of this statement, the identifier is assigned the address of the first byte of this allocated storage.

When you use a DS directive inside a DSECT, the Assembler does not actually place any code in the object file. Using DS statements within a DSECT is an easy way to define a data structure that can later be modified and the program reassembled without affecting other assembly statements in the source program.

## **MSB**

Form: MSB ON or MSB OFF

The Most Significant Bit directive (MSB) provides a means of controlling the value of the most significant bit (msb) of the ASCII characters that are generated by the Assembler. You can use this directive as many times as necessary in your assembly program. The ASCII characters affected by MSB are those generated as immediate string constants and as the string operand of the ASC directive, but *not* of the DCI directive.

The Assembler default is MSB ON because the APPLE II Monitor routine COUT expects ASCII characters in this form for normal display.

### **ASC**

Form: ASC .string.

The ASCII (ASC) directive defines a string of 8-bit bytes in the output object file that are filled with the ASCII values of the characters in the operand string of the directive. Four or fewer bytes are printed on each source line—without a line number—if the LST Gen option is in effect for the directive. When the LST NOGen option, the default, is in effect, only one line and the first four bytes are printed. If a label is present on the ASC directive, it is assigned the current value of the program counter—the address in memory of the first character of the string constant.

The operand string (shown above as .string.) is a delimited string that begins with any character that does not occur in the string and is optionally terminated with the same delimiter. You can omit the terminating delimiter if you also omit the comment field in the assembly statement. The MSB directive determines if the most significant bit of each character in the generated bytes is a one or a zero.

### **STR**

Form: STR .string.

The String (STR) directive provides a easy method of creating a string of ASCII characters preceded by a count byte. The count byte contains the number of characters in the string, not including the count byte itself. The MSB directive determines if the most significant bit of each character in the generated bytes is a one or a zero.

### **DCI**

Form: DCI .string.

The DCI directive functions just like the ASC directive, except that the MSB directive does not control the most significant bit of each byte. Instead, the Assembler generates all bytes of the DCI string with a most significant bit of zero, except for the last byte, which has a most significant bit of one.

## DATE

Form: DATE

This directive generates nine bytes of ASCII data in the output file. These bytes occupy locations \$3B8 thru \$3C0 that normally contain the date portion of the ASMIDSTAMP file that the Editor loaded when you first ran the Editor/Assembler program. This date is typically a sequence of ASCII characters in the DD-MMM-YY format. If the Editor did not find the ASMIDSTAMP file when you initially ran the Editor/Assembler program, the Date directive generates nine ASCII nulls with the most significant bit on.

## IDNUM

Form: IDNUM

This directive generates six bytes of ASCII data, that occupy locations \$3C3 through \$3C8, in the output file. These locations normally contain the six-digit ASCII number portion, with the most significant bit on, of the ASMIDSTAMP file. Since the Editor increments this number each time you type the ASM command from the Editor command level, it provides a way to track the latest assembly. Since this information is printed in the header line of the output assembly listing, this directive also provides a means of marking an object module so you can associate it with a particular printed assembly listing.

**Conditional assembly:** a process that lets you assemble only selected portions of your program so the same source program can be used to generate several executable versions.

A **conditional assembly block** is a group of source statements that are assembled only if certain conditions are met.

---

## Controlling Conditional Assembly

The Assembler recognizes a number of Assembly directives that let you control the sections of a program source file that you want to include in the assembled object file. This process is known as **conditional assembly**. Conditional assembly is useful when you want to write a single generic program that can later be assembled to run in different environments, such as *production* versus *test*, or *machine configuration x* versus *machine configuration y*.

### DO, IFXX, ELSE, and FIN

All these conditional-assembly directives function together to mark the beginning and end of a section of conditional-assembly source statements called a conditional assembly block.

- The DO directive marks the beginning of a conditional assembly block.
- The FIN directive marks the end of a conditional assembly block.

- The ELSE directive divides the statements between the DO and FIN directives into two conditional assembly blocks; the second conditional block is an alternate block that is assembled only if the first block is not.

Use the other conditional-assembly directives (IFNE, IFEQ, IFLT, IFLE, IFGT, IFGE) in the same way that you use the DO directive. These directives are referred to as IFxx directives in the following discussion. Always follow them later in your source program by the FIN directive.

The format of a conditional assembly block is:

DO expression or IFxx expression

( conditional block #1 )

[ ELSE

( conditional block #2 ) ]

FIN

Items within square brackets [ ] are optional.

The DO or IFxx directives mark the beginning of a conditional block of source statements. When the Assembler encounters a DO or IFxx directive, it evaluates the operand expression and compares the result to the value zero. You must previously define in your source file any identifiers you use in this operand expression.

The DO and IFxx directives control the assembly of a conditional block. Assembly of the statements in the block occurs when:

Directive	Resulting Expression
DO	<> 0
IFNE	<> 0
IFEQ	= 0
IFLT	< 0
IFLE	<= 0
IFGT	> 0
IFGE	>= 0

If the value of the operand expression does not cause assembly of the block, the Assembler indicates the source statements it ignores by placing an *S* where the address is normally listed.

If you include an ELSE directive between a set of DO and FIN directives, you define the start of a second or alternate conditional assembly block. This alternate block is assembled only if the first conditional block is *not* assembled. The ELSE directive marks the end of the first block and the start of the alternate block. You can use an ELSE directive only within the range of a DO-FIN pair.

The FIN directive marks the end of the entire conditional assembly block. When the Assembler encounters a FIN directive, it assembles all subsequent statements in your source file in a normal manner.

The Assembler does not recognize assembly variables that you can change within your source program. However, you can use dummy identifiers that you define along with these conditional assembly directives to control a variety of different assembly conditions.

This example demonstrates how you can use the conditional assembly.

```
IFLE TABSIZE=356      ; DOES TABLE FIT
                       ; IN ONE PAGE?
DS 356                 ; IF SO, USE
                       ONE-PAGE TABLE
ELSE                   ; OTHERWISE ...
DS 512                 ; USE TWO-PAGE TABLE
FIN                   ; END OF
                       CONDITIONAL BLOCKS
```

#### **FAIL**

Form: FAIL pass,,string.

You can use the FAIL directive in conjunction with the conditional assembly directives to provide a programmer error-message facility. When the Assembler encounters this directive while assembling your source file, it prints the message contained in the delimited string. Typically, you enclose the Fail directive inside a conditional assembly block that is assembled only when an error condition of some type is detected.

The pass expression in the operand must be a number: 1, 2, or 3. This value is the Assembly pass number in which the Assembler prints the FAIL error message, with 3 meaning both pass 1 and pass 2 failed. The .string. is the message text; it must be enclosed within delimiters like an ASC operand.

This directive is useful for automatically performing size checking on the matching halves of data tables, checking for code page alignment, and similar cross checks between things that must match in size or have particular relationships. You can use any of the conditional assembly directives with this FAIL directive to generate these automatic warnings.

---

### **Source File Directives**

You can use the source file directives to control the source files that the Assembler assembles, and to control the size of the buffers that the Assembler uses to read these files.

#### **CHN**

Form: CHN filename[.slot] [drive] [vol]]

The Chain (CHN) directive is used to connect the segments of a large source program. The slot, drive, and volume expressions are optional; the filename is required. All statements following a CHN directive are ignored; use the CHN directive only as the last statement of a source file. If the filename that you include in the operand field of this assembly statement does not correspond to any file that exists on the indicated disk, the Assembler signals a FILE NOT FOUND error and aborts the assembly. If you indicate the number of a nonexistent disk drive or an empty disk slot, the Assembler signals an I/O ERROR message and aborts your assembly.

If you use a volume parameter and you are using Disk II drives, the Assembler checks the volume number of the disk in the indicated drive. If the volume number doesn't match the one you specified, the Assembler prompts you for the correct volume. This feature lets you assemble large programs requiring multiple disks using only a two-drive system. The Assembler requires that one disk be available at all times to store the object file that it generates.

To assemble multi-disk source files, you must have previously used the DOS INIT command to give your source disks different volume numbers. You can then use these volume numbers in the CHN or INCLUDE directives in your program source file.



## **INCLUDE**

Form: INCLUDE filename[.slot] [drive] [volume]]

The INCLUDE directive is used for one level of source file nesting. When the Assembler encounters this directive, it suspends assembling statements from the current source file and starts reading and assembling the statements in the file named in the operand field. Various errors are given if the file cannot be found. The Assembler prompts you to mount the proper volume if it detects a volume mismatch. This volume-switching arrangement is described in the section on the CHN directive.

You cannot use an INCLUDE directive in an included file; that is, the Assembler does not permit nested include files. You also cannot use the INCLUDE directive in your source programs when you are using coresident assembly, that is, when the Assembler takes your source program from memory rather than from disk.

## **SBUFSIZ and IBUFSIZ**

Form: IBUFSIZ expression

SBUFSIZ expression

These two directives let you optimize the size of the buffers the Assembler uses when reading your source and include files during assemblies. The default size of the source buffer is four pages or 1024 bytes; that of the include buffer, 16 pages or 4096 bytes. These defaults are just about optimal for doing multi-file assemblies using include files with Disk II drives.

The Assembler prints a FREE SPACE PAGE COUNT message at the end of your assembly listing. This count is the number of memory pages that were not used by the Assembler for symbol and/or RLD tables at the end of your assembly. You can use this information in conjunction with the IBUFSIZ and SBUFSIZ directives to increase the size of the source and include buffers to speed printing and reduce assembly time.

Be careful not to make the buffers too large when using Disk II drives; If you use a 32 page or larger buffer, the disk drive motor may timeout and stop spinning between disk reads, resulting in an assembly time that is actually slower. You can use larger buffers more effectively when you are using fast disks.



The Assembler evaluates the operand expression in two ways:

- If the value is less than 128, the Assembler takes the value to be the size of the buffer in pages, where a page is 256 bytes.
- If the value is greater than 256, the Assembler takes the value to be the size of the buffer in bytes. If the value is not divisible by 256, the Assembler truncates the buffer size to the nearest whole number of pages.

If the resultant number of pages is greater than 127, the Assembler signals an **OVERFLOW** error. If the buffer size exceeds available memory, the assembly is aborted. This also happens when the new size of **SBUFSIZ** shrinks the source buffer, which contains the current source file, so small that the source file would be truncated.

You can use these directives only within a source file; if you use them in an include file or in a macro definition file, the Assembler displays an **INVALID FROM INCLUDE/MACRO** error.

### **MACLIB**

Form: **MACLIB** [slot [,drive] [,volume] ] ]

The Assembler supports a simple macro capability with disk-based macros. Since the DOS 3.3 file structure does not have a library file type, macros can be supported only as individual files. This directive allows you to use the macro capability. If you attempt to use macros without the **MACLIB** directive, an **UNKNOWN OP CODE** error message occurs for all macro references. In-line macros are not supported.

The parameters are all optional, and if you do not supply any, macros are read from the first source file's slot, drive, and volume. Each macro must be a separate file in the catalog of the disk being used for macro storage. Assembling disk-based macros is slow, because the parameter-string substitution process requires scanning every character of the macro source.

You can call macros by using a mnemonic that is not in the Assembler's mnemonic table. When the Assembler cannot find a match in its table, it then attempts to open a DOS text file with the mnemonic as the filename and the slot, drive, and volume

Detailed use of macros and arguments: see the section "Using Macros" in this chapter.

parameters that you gave in the MACLIB directive. If the file does not exist, the assembly is aborted. When the text file is found, the source statements from the macro file are assembled using the argument strings to perform string substitution on the dummy arguments.

Since the Assembler searches its mnemonic table first, you cannot use a macro name that is already the name of an existing 6502 instruction mnemonic or an Assembly directive. Macro names must be both valid Assembler identifiers (that is, no space characters) and valid DOS 3.3 filenames.

---

### ***Controlling Your Assembly Listings***

You can use the directives and directive options described in this section to control the format and presentation of the assembly listings generated by the Assembler. These directives are entirely optional; you need not use any of them in your program source files to produce executable programs. Using them improves the quality of your printed assembly listings, and at the same time saves space in your program source files.

You can include an identifier in the label field of any assembly statement containing a listing directive, but this is not recommended. Because assembly statements that contain listing directives are not printed in assembly listings, defining identifiers in this manner can result in incomplete documentation of your program.

#### **PAGE**

Form: PAGE

You can use the Page directive to send an ASCII form feed character to the output device, thus causing a page eject. It also sends a blank line to the Apple II video screen. The Page directive itself does not print as a line on the listing, but its presence is shown by the action and the *missing* line number in the listing. When you are using the Editor, you can find this line by searching for the location of the missing line number.

### **LST**

Form: LST (ON , OFF) [, [NO]option [, [NO]option] ...]

or

LST [NO]option [, [NO]option] ...]

where options are defined by the first letter shown:

Option	Name
Cyc	Cycle times
Gen	Generated object code
Warn	Warnings
Unasm	Unassembled source
Exp	Expanded macro source
Asym	Alphabetic symbol table
Vsym	Value-ordered symbol table
Sixup	Sixup symbol table

The Listing (LST) directive lets you suppress all or part of the source listing. Turning the listing off by using the LST OFF directive increases the speed of your assembly; this is most noticeable when you are doing a large assembly and listing it to a printer. You can use this directive any number of times to turn on and turn off selected parts of your program's listing.

The options give you additional control of the information in the listing file. You can use any number of these options with or without the ON/OFF print control. You need to specify only the first letter of each option. If you do not use the LST directive in your source program or do not specify any options, the Assembler assumes that you want

LST ON, NOCyc, NOGen, Warn, Unasm, Exp, Asym, NOVsym, NOSixup

### **Cycle Times**

Form: NOCyc (OFF) (default)

The Cycle Times option lists the 6502 instruction cycle times for each assembly instruction in your source file. These times are printed in the listing as three extra columns just to the left of the source line numbers. The cycle time for each instruction is printed as a single digit within parentheses.

The Assembler does not normally print instruction cycle times unless you specify the Cyc option. Typically, you do not want to have the cycle times printed unless you are programming timing-sensitive code.

### **Generated Object Code**

Form: NOGen (OFF) (default)

You can use the Gen option to control the printing of all the object code bytes generated by the data directives if this data requires more than one print line. This option does not affect what the data directives generate, only what they print.

The data directives; see the section  
"Generating Data in Your Object  
Code."

Unless you specify the Gen option, the Assembler normally lists only the first four bytes of object code generated by a data directive. The Assembler's default state is NOGen, since this saves printing time.

### **Warnings**

Form: Warn (ON) (default)

This option turns printing of Assembler warnings on or off. The Assembler normally prints all warnings it encounters as it assembles your file. You can suppress warnings by using the NOWarn option. The Assembler prints a warning total, and also an error message total, at the end of the listing even if you use the NOWarn option.

### **Unassembled Source**

Form: Unasm (ON) (default)

You can use the Unassembled Source option to suppress printing of statements in your program that are not assembled because they are contained in a conditional assembly block. Normally, the Assembler prints all the statements in your source program and marks those statements that are unassembled.

### **Expanded Macro Source**

Form: Exp (ON) (default)

You can control the printing of macro source statements using this option. The Assembler normally prints all the statements in a macro definition whenever it encounters a macro in your source program. Use the NOExpand option to suppress the printing of macro expansions. You can use this option within the macro definition itself if you want always to suppress this printing. This option does not affect macro operation in any way, just the printing of source statements in the listing file.

### **Alphabetic Symbol Table**

Form: Asym (ON) (default)

You can turn off the Assembler's normal symbol-table-dump listing using the NOAsym option. The Assembler normally prints the symbol table in alphabetical order unless you disable it using the NOAsym option.

### **Value-Ordered Symbol Table**

Form: NOVsym (ON) (default)

If you select the Vsym option, the Assembler prints the symbol table by order of the symbol values. The Assembler requires an extra two bytes per symbol over the size of the symbol table at the end of pass 2 to complete this successfully. When the available memory is not sufficient, the Assembler uses what is available and prints only the symbols that it can sort. If you have a large assembly that uses up most of the 24K of symbol table space, the value-ordered symbol table may not contain all the identifiers.

### **Sixup Symbol Dump**

Form: NOSixup (OFF) (default)

When you direct the listing to a printer, this option causes the symbol table dump to be printed in six columns, instead of the normal four. This assumes that your printer can print 120 or more characters per line. The four-column default results in a table that is 80 characters wide.

### **REP**

Form: REP expression

You can use the Repeat (REP) directive to print a string of characters in your listings, starting at the first character of the source statement portion of the listing. The default character that is printed is the asterisk ( \* ): you can change this character by using the CHR directive. You can use any number of REP directives in your source file.

You use this directive to print a string of asterisks to set off comment headings at the beginning of subroutines or modules, or otherwise to make your listings more readable. By using this directive instead of simply inserting the string of asterisks in your file, you can save considerable space in your source file.

Only the low byte of the expression is used. You can specify that from one to 256 of the currently defined CHR's are to be printed. If you specify 0 or 256, 256 characters will be printed.

### **CHR**

Form: CHR /?

Use the Character (CHR) directive to change the character repeated by the REP directive.

The ? represents the character you want to see printed by the REP directive. You must precede this character by a delimiter, shown here as a slash. Both the printed character and the delimiter can be any character. The CHR directive can be used any number of times in your source file to change the printed character in different parts of your listing.



### ***SKP***

Form: SKP expression

The Skip (SKP) directive lets you insert a number of blank lines in the listing by sending ASCII carriage returns to the output device. The device must provide its own line feed after a carriage return if the device requires a line feed to advance a print line on the paper.

### ***SBTL***

Form: SBTL .string.

The Subtitle (SBTL) directive lets you place a title line (specified by the delimited string shown above as .string.) at the top of each page of the listing file. Using SBTL is optional, but it provides a means of identifying a specific listing.

When you use this directive, the first line of each page contains the current subtitle followed by the Assembler ID Stamp, which is the date followed by a six-digit integer. The Assembler prints blanks in the title line if the ASMIDSTAMP file is not available for an assembly.

## ***Using Macros in Your Assembly-Language Programs***

The MACLIB directive described in the section "Source File Directives" lets you use the disk-based macro capability of the Assembler. The Assembler supports macro definitions that you create prior to the assembly and store on a disk as a separate DOS text file. You cannot define macros within your program source file.

When you use the MACLIB directive in your program source file, you tell the Assembler that you may use macros later in this file. The MACLIB directive also tells the Assembler on which disk the macro definitions are stored. After you use the MACLIB directive, the Assembler assumes that any mnemonic you use in your source file that is not a standard 6502 instruction or Assembly directive mnemonic is the name of a macro definition that resides on this disk.

---

### **Calling Macros in Your Program Source File**

To call a macro in your program source file, include the name of the macro in the mnemonic field of an assembly statement, just as you would use any other mnemonic. This macro name must be the name of the DOS text file that resides on the disk volume you specified in your earlier MACLIB directive. You can include operand expressions in the operand field of these assembly statements, but you cannot include a comment.

When the Assembler encounters a mnemonic that does not match either a standard 6502 instruction, SWEET16 mnemonic, or Assembly directive, the Assembler suspends its assembly of the current source file, preserves any necessary information, and attempts to read a file from the macro source disk having the mnemonic name as the filename. If this macro definition file doesn't exist, the Assembler signals the error and *aborts* the assembly.

---

### **The Macro Definition File**

The macro definition file is a text file consisting of regular 6502 assembly statements. There are no special macro directives that you must use in defining a macro.

When you call a macro in your program source file, you can specify up to nine string parameters in the operand field of the statement containing the macro name. You must separate each of these string parameters by a comma. The Assembler parses this operand field using the comma as a delimiter and substitutes these string parameters into your macro definition wherever you indicated. Two commas with no characters in-between are assumed to represent a null parameter. You cannot pass a comma as part of a string parameter except as a numeric constant, that is, as \$2C or \$AC.

In the macro definition file that you created on the macro source disk, you can indicate where these string parameters are to be substituted by using the two-character sequences &1 through &9. The Assembler replaces these sequences with the characters of the first through ninth string parameters, respectively. If you reference a parameter in your macro definition that you did not supply when you called the macro in your source file, the Assembler replaces that parameter with zero (no) characters. You can use any number of &n parameters within a macro statement or within a single field of a statement.

An example of a macro definition is:

```
LDA    &1
CLC
ADC     &2
STA     &3
LDA     &1+1
ADC     &2+1
STA     &3+1
```

If you save this example macro definition in a file with the name ADD16, you can call the macro as shown:

```
MACLIB      ; ENABLE MACROS
VALU1       EQU    $FA
VALU2       EQU    $FB
SUM         EQU    $FC

; INVOKE ADD16 MACRO BELOW
ADD16      VALU1, VALU2, SUM
```

When you assemble this source program, the Assembler expands this macro and substitutes the three parameters into the macro definition where you indicated. This macro expansion appears in your assembly listings as:

```
1 MACLIB ; ENABLE MACROS
2 VALU1   EQU    $FA
3 VALU2   EQU    $FB
4 SUM     EQU    $FC
5 *****

10 ; INVOKE ADD16 MACRO BELOW
11     ADD16  VALU1, VALU2, SUM
1+     LDA    VALU1
2+     CLC
3+     ADC    VALU2
4+     STA    SUM
5+     LDA    VALU1+1
6+     ADC    VALU2+1
7+     STA    SUM+1
```

The Assembler always lists the macro expansion following the assembly statement that calls the macro (line 11). The lines of a macro are indicated in the assembly listing by the plus character following the line number. You can suppress the expansion of a macro by using the LST NOEXP directive described in the section "Controlling Your Assembly Listings."

Notice that the Assembler substituted the `&1` argument with the first string parameter, `VALU1`, and so on for all of the macro parameters. The process of substituting macro parameters must not produce an assembly statement longer than 255 characters. This can happen if you attempt to insert a large number of long string parameters into a long macro statement. If you use a macro statement with a long comment field, you may also encounter this problem.



#### **Warning**

Do not place a comment after a macro string parameter. If you do, the Assembler tries to assemble the comment.

The Assembler supports two special features to help you use macros. These features are the `&0` and `&X` parameters. The following paragraphs describe how you can use these parameters inside your macro definitions.

#### **The `&0` Parameter**

You can use the `&0` parameter in your macro definition to represent the number of parameters that are present in the assembly statement that calls this macro. The Assembler always counts the number of parameters present in the operand field of the statement calling the macro. It then substitutes this single-digit number wherever it finds the `&0` parameter in your macro definition.

You can use this `&0` parameter within a conditional assembly statement in your macro definition, either to validate the macro call or to create flexible macro definitions.

The `&0` parameter represents the number of parameters that are present in the assembly statement calling a macro.

The **&X** parameter represents the number of times macros are used during the assembly. It is useful for generating unique identifiers within a macro definition.

### **The &X Parameter**

You can use the **&X** parameter (*always* uppercase X) in your macro definitions to represent a cumulative count of the number of times that you used any macro during this assembly. The Assembler substitutes a one- to four-digit numeric string for each occurrence of the **&X** parameter, starting initially with the string 1, and increasing this number by one each time you call a macro during your assembly.

The **&X** parameter lets you automatically generate unique labels within a macro expansion, even if you call the macro definition many times. All the labels you generate using the **&X** parameter appear as individual entries in the symbol table and all appear in the symbol table dump.

You can use the **&X** parameter in your macro definition by appending it to some label prefix or embedding it inside a label, as shown in this example:

```
1      LDY #5
2 F&XL  STA &1,Y
3      DEY
4      BNE F&XL
```

This example shows **&X** embedded in a label. The Assembler will create the unique labels F1L, F2L, and F29L when this macro is used as the first, second and twenty-ninth macros of an assembly.





## ***The Bugbyter Debugger***

---

<b>106</b>	<b>Introduction to Bugbyter</b>
<b>107</b>	<b>Restrictions on Using Bugbyter</b>
<b>108</b>	<b>A Tutorial: Using Bugbyter</b>
<b>109</b>	<b>Getting Started</b>
<b>110</b>	The Register Subdisplay
<b>111</b>	The Stack Subdisplay
<b>112</b>	The Code Disassembly Subdisplay
<b>113</b>	The Memory Cell Subdisplay
<b>113</b>	The Breakpoint Subdisplay
<b>114</b>	The Bugbyter Command Line
<b>116</b>	Typing Bugbyter Commands
<b>117</b>	Loading Your Program
<b>119</b>	Single-Stepping Through Your Program
<b>122</b>	Using the Memory Subdisplay
<b>125</b>	Tracing Your Program
<b>126</b>	Changing Your Program in Memory
<b>127</b>	Viewing a Page of Memory
<b>129</b>	<b>Using Bugbyter</b>
<b>130</b>	Relocating the Bugbyter Program
<b>130</b>	Using Bugbyter in a Language Card
<b>131</b>	Entering the Monitor
<b>131</b>	Restarting Bugbyter
<b>132</b>	<b>Memory and the Bugbyter Displays</b>
<b>132</b>	Using the Memory Subdisplay
<b>133</b>	Viewing the Memory Page Display
<b>135</b>	Altering the Contents of Memory
<b>136</b>	Altering the Contents of Registers
<b>137</b>	Altering Bugbyter's Master Display Layout

<b>139</b>	<b>Controlling the Execution of Your Program</b>
<b>139</b>	Using the Single-Step and Trace Modes
<b>141</b>	Single-Stepping Your Program
<b>141</b>	Using Trace Mode
<b>142</b>	Tracing Subroutines
<b>142</b>	Setting Transparent Breakpoints
<b>144</b>	Using Breakpoints
<b>144</b>	Clearing Transparent Breakpoints
<b>145</b>	Adjusting the Trace Rate
<b>145</b>	Using Display Options in Trace and Single-Step Mode
<b>148</b>	Using Execution Mode
<b>148</b>	Setting Real Breakpoints
<b>149</b>	Debugging Your Program in Execution Mode
<b>150</b>	Debugging Real-Time Code
<b>152</b>	Debugging Programs that Use the Keyboard and Display
<b>153</b>	Eliminating Screen Contention
<b>153</b>	Eliminating Keyboard Contention
<b>154</b>	Using Hand Control 0 to Control Trace Mode
<b>155</b>	Using Hand Control 0
<b>156</b>	Executing Undefined Opcodes

## The Bugbyter Debugger

Every assembly-language programmer eventually runs up against a newly-written or revised program that just doesn't work as intended. This isn't any reflection on you as a programmer; everyone occasionally overlooks a problem condition or omits an essential instruction, resulting in a program *bug*.

The origin of the term **bug**, meaning a computer error, can be traced back to the origin of computers themselves.

According to U.S. Navy Captain Grace Murray Hopper, a pioneer in computer technology during World War II, the first computer *bug* was discovered at Harvard in August 1945. Hopper, interviewed at age 74, recalled that at the time she and her associates were working on the Mark I, which she affectionately called "the granddaddy of today's computers:"

"Things were going badly; there was something wrong in one of the circuits of the long, glass-enclosed computer. Finally, someone located the trouble spot and, using ordinary tweezers, removed the problem—a two-inch moth. From then on, when anything went wrong with a computer, we said it had bugs in it."

A real test of your programming skill is how quickly you can locate problems in your programs and fix any errors to produce a working program. With the proper tools, this process of testing and fixing your program is easy and efficient, allowing you to produce error-free quality software.

The process of testing and changing a program to eliminate errors is called **debugging**.

The Bugbyter program is a display-oriented debugging tool that can save you considerable time in testing and debugging your assembly-language programs. Using Bugbyter for just a few minutes, you can see precisely how your program is executing and where things are going wrong. The Bugbyter program is also a useful tool for testing and verifying a working program to make sure that it operates correctly under a variety of conditions.

Bugbyter functions and commands: see Appendix C.

This chapter describes how you can use Bugbyter to test and debug your assembly-language programs:

- It describes the characteristics of the Bugbyter program and how Bugbyter can help you to debug your programs.
- It contains a brief tutorial on using Bugbyter to test the assembly-language program you created in the tutorials in Chapters 2 and 3.
- It describes how you can use all Bugbyter's features while testing and debugging your programs.

### ***Introduction to Bugbyter***

The Bugbyter program lets you load and control the execution of your assembly-language program on any Apple II system. You can use Bugbyter to test and debug almost any assembly-language program as long as your program leaves enough storage space to hold the Bugbyter program somewhere in your Apple II's memory. The Bugbyter program can be loaded and executed almost anywhere in memory, allowing you the greatest flexibility in debugging your assembly-language programs.

This ability to relocate Bugbyter is discussed later in this chapter.

Bugbyter gives you a variety of options to use when debugging your 6502 assembly-language programs. At any time you can view the status of the 6502 registers, stack, and memory as they appear to your assembly-language program. When you are debugging internal portions of your program and do not need to view your programmed displays, you can use Bugbyter's Master Display to show you

- all of the Apple II's 6502 internal registers
- a portion of the Apple II's program stack
- a mnemonic disassembly of portions of your assembly-language program
- portions of your computer's memory
- any real or transparent breakpoints that you set.

Using Bugbyter, you can step through your program one instruction at a time, observing all the effects of executing each instruction. If you want, you can alter the layout of Bugbyter's Master Display to show you more or fewer stack or memory locations, program statements, or breakpoints.

At any time, you can change the conditions under which you are testing or debugging your program. For example, you can

- change the contents of any 6502 register;
- alter the contents of memory locations or the stack to change data values that may be stored there;
- alter instructions or parts of your program and immediately test these revisions.

To fix or change quickly the 6502 assembly-language instructions that comprise your program, you can enter 6502 assembly-language mnemonics directly into your Apple II's memory. Bugbyter translates these instructions and stores the actual machine codes into the memory locations that you select.

You can indicate to Bugbyter regions of code that must execute in real-time. This lets you test programs that make use of some of the Apple II's real-time operating system routines, and also test your own programs that contain portions that must execute in real-time. Bugbyter tests these programs using all its debugging facilities, while still allowing execution at full speed.

---

### ***Restrictions on Using Bugbyter***

You can use Bugbyter to test and debug any assembly-language program as long as your own program leaves a contiguous block of memory 6700 (\$1A00) bytes long somewhere in your Apple II's memory. This memory is needed to contain the Bugbyter program code and data areas. Bugbyter also uses the first 32 (\$20) bytes of the Apple II's stack (memory locations \$100 to \$11F); this should not cause problems unless your own program alters the contents of the beginning of the stack.

The section "Relocating the Bugbyter Program" in this chapter explains how to relocate Bugbyter to prevent conflicts with your own program.

Allowing for these memory restrictions, there are that you cannot use Bugbyter to test and debug. your Apple IIe system use bank-switching to external memory, Bugbyter must remain resident in memory. Although you can debug programs that use bank-switching, you cannot swap Bugbyter out of usable memory space for debugging.

### **A Tutorial: Using Bugbyter**

In the tutorial that follows, you use Bugbyter to test the short program you created and assembled in Chapters 2 and 3. It shows you how to test an assembly program to verify that it works as you intended and how to modify your programs so they execute differently.

To work through this tutorial, you need

- your Apple II system
- the DOS Programmer's Tool Kit Volume II disk containing the Bugbyter program and your TESTPROGRAM.C that you created in the tutorial in Chapter 3.

In this tutorial, you will

- become familiar with Bugbyter's Master display
- load your assembly-language object program
- test your program using the Single-Step and Trace modes;
- change your program and test it again.



## Getting Started

### What You Do

1. Insert the DOS Programmer's Tool Kit Volume II disk into your disk drive 1 and turn on your Apple II. After your Apple II loads the operating system from the disk and displays the Applesoft prompt (|), type

```
BRUN BUGBYTER
```

and press **RETURN**.

### What Happens

Your Apple II loads the Bugbyter program.

Bugbyter then shows you its Master Display on your video screen. This Master Display is divided into six subdisplays (Figure 4-1).

**Figure 4-1.** Bugbyter's Master Display

① Register subdisplay: 6502 and Bugbyter registers

② 6502 Stack with Stack Pointer highlighted

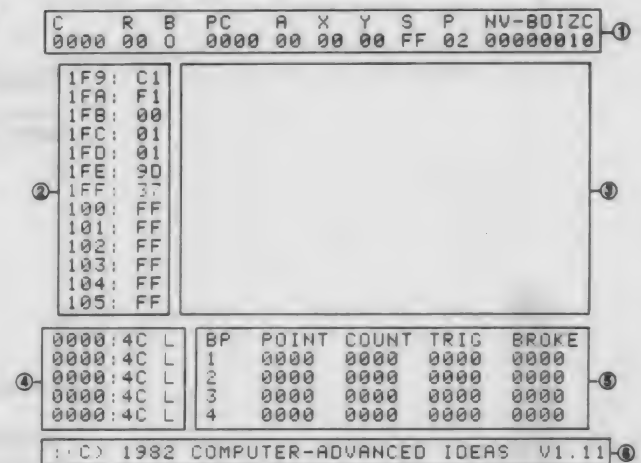
③ Code Disassembly display

④ Memory Cell subdisplay

⑤ Breakpoint subdisplay

⑥ Bugbyter Command Line

Your screen may appear with slight differences.



### The Register Subdisplay

The six 6502 registers and three Bugbyter registers are shown in the Register subdisplay at the top of the screen (Figure 4-2).

Figure 4-2. The Register Subdisplay

C	R	B	PC	A	X	Y	S	P	NV-BDIZC
0000	00	0	0000	00	00	00	FF	02	00000010

1F9	C1
1FA	F1
1FB	00
1FC	01
1FD	01
1FE	00
1FF	3F
100	00
101	00
102	00
103	00
104	00
105	00

0000	40	L	BF	POINT	COUNT	TRIG	8F00E
0000	40	L	1	0000	0000	0000	0000
0000	40	L	2	0000	0000	0000	0000
0000	40	L	3	0000	0000	0000	0000
0000	40	L	4	0000	0000	0000	0000

PC 1992 COMPUTER-ADVANCED IDEAS 01 11

The 6502 registers are:

6502 Register	Name
PC	Program Counter
A	A-Register
X	X-Register
Y	Y-Register
S	Stack Pointer
P	Processor Status Register

The display in the upper-right corner shows the Processor Status Register in both two-digit hexadecimal (P) and in binary (NV-BDIZC) notation, where the individual flags are:

Flag	Name
N	Negative
V	Overflow
B	Break

Flag	Name
D	Decimal
I	Interrupt
Z	Zero
C	Carry

The Bugbyter registers in the upper-left corner of the display are explained later in this chapter; they are:

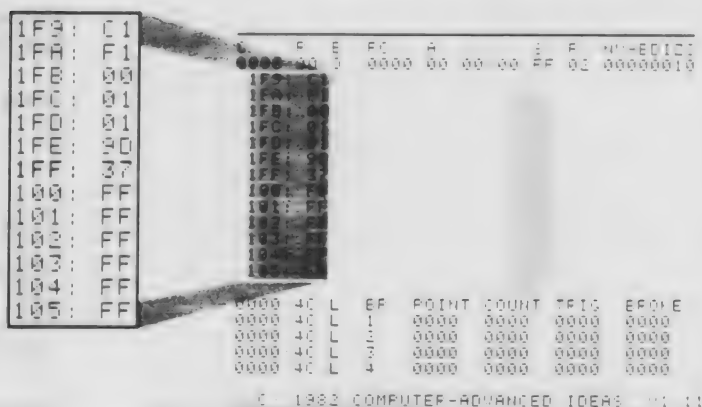
Bugbyter Registers	Name
C	Cycle Count
R	Trace Rate
B	Breakpoint Flag

The breakpoint flag is either O (out) or I (in).

### The Stack Subdisplay

Bugbyter's Stack subdisplay (Figure 4-3) acts as a window into the 6502 memory stack. The Stack subdisplay contains the ascending addresses of memory locations just before and after the location pointed to by the Stack Pointer, and thus shows the contents of each byte in a portion of the stack.

Figure 4-3. The Stack Subdisplay



**Figure 4-4.** The Code Disassembly Subdisplay

: (C) 1982 COMPUTER-ADVANCED IDEAS 01.11

These machine-instruction bytes (A0 C0) are displayed as one of Bugbyter's information display options. As you trace or single-step through your program, you can select one of seven display options.

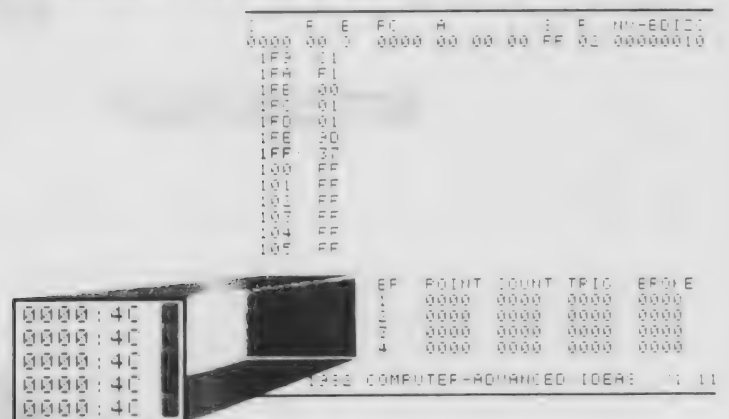
112

MEM command: see the section  
"Viewing and Altering Memory."

**Figure 4-5.** The Memory Cell  
Subdisplay

### The Memory Cell Subdisplay

The Memory Cell subdisplay (Figure 4-5) shows the contents of a number of memory locations that may be important to your program. You can use the MEM command to select the addresses of individual bytes or byte-pairs that Bugbyter displays continuously in this portion of the Master Display.

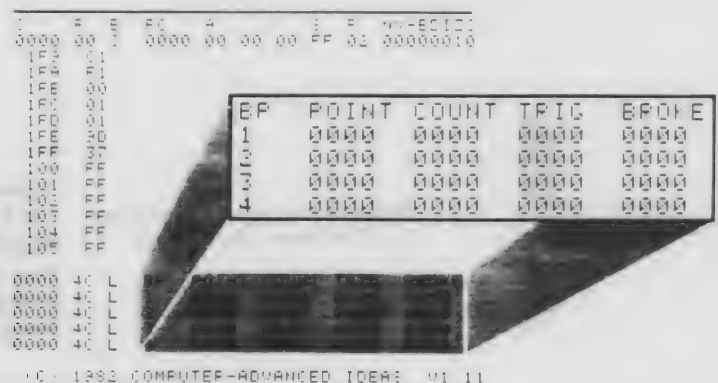


Breakpoints and other techniques that  
you can use in debugging: see the  
section "Controlling the Execution of  
Your Program."

**Figure 4-6.** The Breakpoint Subdisplay

### The Breakpoint Subdisplay

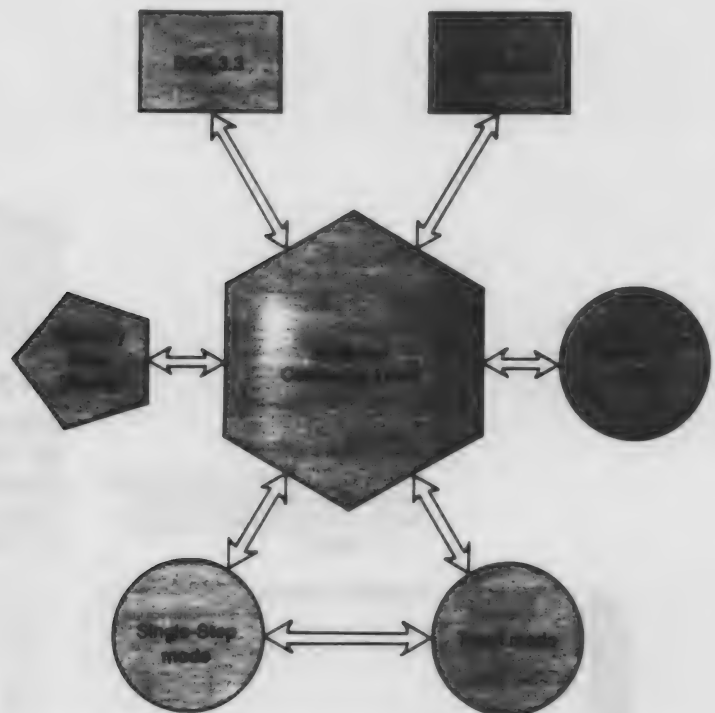
Bugbyter allows you to set a number of program breakpoints that you can use to control the execution of your program. The Breakpoint subdisplay area (Figure 4-6) of the Master Display, just below the Code Disassembly subdisplay, shows these breakpoints and relevant breakpoint information.



### **The Bugbyter Command Line**

On the last line of the display, you see the Bugbyter command line, which currently shows a brief copyright notice. The blinking cursor just after the colon (:) prompt indicates that Bugbyter is ready to accept commands from you.

**Figure 4-7.** Bugbyter Command Line Allows You Access to All Features



The Bugbyter command line is the last line on the Master Display screen.

When you first start up the Bugbyter program, you are placed at the Bugbyter command level. From this level, you can access all Bugbyter's features by typing short commands on the command line. You can also use commands to enter one of Bugbyter's three debugging modes to access more features. Figure 4-7 shows the five Bugbyter modes and the things that you can do in each mode.



The things you can do at the **Bugbyter command level** are:

- execute DOS commands, such as BLOADing programs
- view or alter contents of memory by entering hexadecimal (hex) values, character codes, or 6502 assembly code
- change the contents of the 6502 or Bugbyter registers
- enter Applesoft/DOS or the Monitor
- alter the layout of the Bugbyter Master Display
- set and clear breakpoints
- view areas of memory, using Bugbyter's Memory Page Display
- enter one of Bugbyter's three debugging modes.

In **Memory Page Display mode** you can

- view areas of memory;
- alter contents of memory by entering hexadecimal values, character codes, or 6502 assembly code.

In **Single-Step mode** you can

- single-step or execute your program one instruction at a time;
- view Bugbyter's Master Display, or view your Apple II's low-resolution, high-resolution, or normal text screens.

In **Bugbyter Trace mode** you can

- trace your program's execution, updating Bugbyter's Master Display after each instruction, scanning for RTS opcodes or transparent breakpoints;
- view Bugbyter's Master Display, or your Apple II's low-resolution, high-resolution, or normal text screens.

In **Bugbyter Execution mode** you can

- execute your program directly on the 6502, using real breakpoints to control execution.

### **Typing Bugbyter Commands**

When you type commands at the Bugbyter command level, you simply type the characters to form the command and press **(RETURN)** when you are done. If you make a mistake when typing, you can use **(←)** to move the cursor back and then retype the proper characters. To erase the line and start retyping it again, press **(CONTROL)-(X)**.

Bugbyter also has some additional editing commands to save you from retyping. If you type a line of characters on the Bugbyter command line and notice an incorrect character earlier on the line, you can use **(←)** and **(→)** to place the cursor over the incorrect characters. Then

- to replace the character, type a new character over the old one.
- to delete the character altogether, press **(CONTROL)-(D)**.
- to insert additional characters before the current character, press **(CONTROL)-(I)** and type the additional characters; to end the insertion press **(←)**, **(→)**, or **(RETURN)**.
- to enter a *control character* on the command line, if you want to store character-string information into memory, press **(CONTROL)-(C)** before the entering the control character.
- to move the cursor to the beginning of the command line, press **(CONTROL)-(B)**.
- to move to the end of the line, press **(CONTROL)-(N)**.

When you finish editing the line, press **(RETURN)**. When you press **(RETURN)**, Bugbyter accepts all the characters on the command line. If there are unwanted characters to the right of the cursor, delete them by pressing **(CONTROL)-(D)** or replace them with spaces by pressing **(SPACE)**.

**Table 4-1.** Bugbyter Command-Line Editing Functions

Table 4-1 is a summary of the editing functions and the keystrokes that accomplish them.

Editing Function	Keystroke
Move cursor <i>left</i> one character	←
Move cursor <i>right</i> one character	→
Move cursor to <i>beginning</i> of line	CONTROL- <b>B</b>
Move cursor to <i>end</i> of line	CONTROL- <b>N</b>
Delete <i>entire</i> line	CONTROL- <b>X</b>
Delete current character	CONTROL- <b>D</b>
Insert characters	CONTROL- <b>I</b>
Enter a <i>control</i> character	CONTROL- <b>C</b> , any character
Accept the current line	RETURN

### Loading Your Program

To debug your program using Bugbyter, you must first load your program into memory by using the regular DOS BLOAD command.

**By the Way:** When you enter a DOS command, the Bugbyter screen display scrolls up and leaves the cursor below the last line of text. Press RETURN to restore the Bugbyter display.

To call a DOS command while using Bugbyter, type a period (.) before you type the DOS command.

1. Type	After your Apple II finishes loading your
. BLOAD	TESTPROGRAM.OBJ0
TESTPROGRAM.OBJ0	program into memory, you see the blinking cursor.
and press RETURN.	
TESTPROGRAM.OBJ0 is the binary file you created in Chapter 3.	

**Note:** Any DOS errors that occur when you execute a DOS command from within Bugbyter leave you at the BASIC command level. To return to Bugbyter, press RESET, then type

CALL 1016

and press RETURN. You return to Bugbyter and see Bugbyter's Master Display.

If you later run Bugbyter in a Language Card, DOS errors leave you in the Monitor. If this happens, press CONTROL-Y and then RETURN to return to Bugbyter.

2. To return to Bugbyter's Master Display, press **(RETURN)**.

3. To view your program that you just loaded into memory, type

1000L  
and press **(RETURN)**.

Bugbyter fills the Disassembly portion of your screen with the first few instructions of your program, starting with the instruction at address \$1000.

The Bugbyter **L** command loads the Code Disassembly subdisplay with the next several instructions following the address you specify.

Since TESTPROGRAM.OBJ0 is a short program, you can see all of it on the screen.

```

C   R   B   PC   A   X   Y   S   P   NV-BDIZC
0000 00 0   0000 00 00 00 FF 02 00000010

1F9: C1   1000: LDY  #$C0      A0 C0
1FA: F1   1002: LDX  #$00      A2 00
1FB: 00   1004: JSR  $1000     20 00 10
1FC: 01   1007: INX             E8
1FD: 01   1008: CPX  #$05      E0 05
1FE: 9D   100A: BNE  $1004     D0 F8
1FF: 37   100C: RTS             60
100: FF   100D: INY             C8
101: FF   100E: TYA             98
102: FF   100F: STA  $1100,X    90 01 11
103: FF   1012: RTS             60
104: FF   1013: BRK             00
105: FF   1014: BRK             00

```

```

0000:4C L BP POINT COUNT TRIG BROKE
0000:4C L 1 0000 0000 0000 0000
0000:4C L 2 0000 0000 0000 0000
0000:4C L 3 0000 0000 0000 0000
0000:4C L 4 0000 0000 0000 0000

```

:\*

You may see different values in the Stack subdisplay; this is normal and not cause for alarm.

You can now visually check the instructions in your program to verify that your program was assembled correctly (in the tutorial in Chapter 3).

The **S** command places Bugbyter in Single-Step mode.

### Single-Stepping Through Your Program

Bugbyter allows you to watch the execution of your program one instruction at a time, letting you see the results after each instruction.

1. To begin single-stepping your program at address \$1000, type

1000S

and press **RETURN**.

The LDY instruction that appears at address \$1000 is highlighted, and the program counter (PC) on the top line of the screen shows the address 1000.

```

C      R  B  PC  A  X  Y  S  P  NV-BDIZC
0000  00  0  1000  00  00  00  FF  02  00000010

1F9:  C1
1FA:  F1
1FB:  00
1FC:  01
1FD:  01
1FE:  90
1FF:  37
100:  FF
101:  FF
102:  FF
103:  FF      1000: LDY  #$C0
104:  FF      1002: LDX  #$00
105:  FF      1004: JSR  $1000

0000:4C L BP POINT COUNT TRIG BROKE
0000:4C L 1 0000 0000 0000 0000
0000:4C L 2 0000 0000 0000 0000
0000:4C L 3 0000 0000 0000 0000
0000:4C L 4 0000 0000 0000 0000

SINGLE STEP

```

Note that no instructions of your program are executed and none of the registers are changed except the PC register. The first instruction of your program to be executed is to load the Y-register with the value \$C0, as indicated by the highlighted LDY instruction on the Master Display.

Pressing **SPACE** in Single-Step mode executes the next instruction.

2. To execute this LDY instruction, press **SPACE**.

Bugbyter executes this instruction of your program and updates the display.

```

1
C R B PC A X Y S P NV-BDIZC
0000 00 0 1002 00 00 C0 FF 00 10110000

1F9: C1
1FA: F1
1FB: 00
1FC: 01
1FD: 01
1FE: 30
1FF: 37
100: FF
101: FF
102: FF      1000: LDY #$C0      P:1011000
103: FF      1003: LDX #$00
104: FF      1004: JSR $1000
105: FF      1007: INX

0000:4C L BP POINT COUNT TRIG BROKE
0000:4C L 1 0000 0000 0000 0000
0000:4C L 2 0000 0000 0000 0000
0000:4C L 3 0000 0000 0000 0000
0000:4C L 4 0000 0000 0000 0000

```

SINGLE STEP

Notice that the program counter is incremented to 1002 and the Y-register is loaded with the hex value C0. In the Disassembly subdisplay, the disassembled instructions are shifted and the instruction at address 1002, the next instruction to be executed, is now highlighted. Bugbyter also displays the contents of the processor status register in binary to the right of the instruction that was just executed.

The next highlighted instruction loads the X-register with the value \$00.

3. Press (SPACE) to execute this instruction.

Verify that the X-register now contains the value \$00.

```

C    R    B    PC    A    X    Y    S    P    NW-BDIZC
0000 00 0    1004 00 00 C0 FF 32 00110010

1F9: C1
1FA: F1
1FB: 00
1FC: 01
1FD: 01
1FE: 00
1FF: 37
100: FF
101: FF
102: FF
103: FF
104: FF
105: FF
1000: LDY #$C0
1002: LDX #$00
1004: JSR $1000
1007: INX
1008: CPX #$05
P:10110000
P:00110010

0000:4C L BP POINT COUNT TRIG BROKE
0000:4C L 1 0000 0000 0000 0000
0000:4C L 2 0000 0000 0000 0000
0000:4C L 3 0000 0000 0000 0000
0000:4C L 4 0000 0000 0000 0000

```

SINGLE STEP

The next instruction is a jump-to-subroutine instruction that calls the Store subroutine at address 100D. This JSR instruction changes the contents of the stack pointer as well as the program counter.

4. Press (SPACE) to execute this instruction.

Bugbyter executes this instruction and calls the subroutine at 100D. Notice that the program counter (PC) contains the address 100D.

Notice that the stack pointer (S) changed to FD and that the stack subdisplay shifted down. The stack pointer points to the location identified by the highlighted bar.



```

C      R  B  PC  A  X  Y  S  P  NV-BDIZC
0000 00 0  1000 00 00 C0 FD 32 00110010

```

```

1F7: 43
1F8: D4
1F9: C1
1FA: F1
1FB: 00
1FC: 01
1FD: 01
1FE: 06      1000: LDY  #$C0      P: 10110000
1FF: 10      1002: LDX  #$00      P: 00110010
100: FF      1004: JSR  $1000      P: 00110010
101: FF      1006: INY
102: FF      100E: TYA
103: FF      100F: STA  $1100,X

```

```

0000:4C L BP POINT COUNT TRIG BROKE
0000:4C L 1 0000 0000 0000 0000
0000:4C L 2 0000 0000 0000 0000
0000:4C L 3 0000 0000 0000 0000
0000:4C L 4 0000 0000 0000 0000

```

SINGLE STEP

5. Execute the next two instructions by pressing **(SPACE)** twice, once for each instruction.

These instructions increment the contents of the Y-register and transfer the contents of the Y-register to the A-register, respectively. You can verify the operation of these instructions by watching the Register subdisplay.

### Using the Memory Subdisplay

The next instruction to be executed is STA \$1100,X; this stores the value in the A-register into memory address \$1100 (if you press **(SPACE)** right now, you will execute this instruction and store the value \$C1 into memory). Before you execute this instruction, use the Memory subdisplay to verify that the proper value gets stored in memory.

Pressing **(ESCAPE)** while in Single-Step mode returns you to the Bugbyter command level.

The **MEM** command lets you modify the Memory subdisplay.

1. Press **(ESCAPE)** to exit from Bugbyter's Single-Step mode and return to the command level.

You see the colon prompt and blinking cursor of the Bugbyter command level.

2. Type  
MEM

The blinking cursor appears in the first position of the Memory subdisplay.

and press **(RETURN)**.

Each cell of the Memory subdisplay consists of an address, followed by a colon and the hex value of the byte stored in that memory location. After the hex value is its corresponding Apple character.

```

C   R   B   PC   A   X   Y   S   P   NV-BDIZC
0000 00 0   100F C1 00 C1 FD B0 10110000

1F7: 43
1F8: 04
1F9: C1
1FA: F1
1FB: 00
1FC: 01      1000: LDY #$C0      P:10110000
1FD: 01      1002: LDX #$00      P:00110010
1FE: 06      1004: JSR $1000     P:00110010
1FF: 10      1000: INY           P:10110000
100: FF      100E: TYA           P:10110000
101: FF      100F: STA $1100,X
102: FF      1012: RTS
103: FF      1013: BRK

0000:4C L BP POINT COUNT TRIG BROKE
0000:4C L 1 0000 0000 0000 0000
0000:4C L 2 0000 0000 0000 0000
0000:4C L 3 0000 0000 0000 0000
0000:4C L 4 0000 0000 0000 0000

```

: MEM

3. To set a particular memory address in this first memory cell, type

1100

and press **RETURN**.

The address 1100 appears in the first memory cell of the Memory subdisplay, followed by the byte currently stored in that memory location. The blinking cursor moves to the next memory cell.

4. To enter the addresses of the next four memory locations, type

1101

1102

1103

1104

You loaded the addresses of five consecutive memory locations into the Memory subdisplay. (As you later execute your test program, your program will fill each of these memory locations with an Apple character code.)

Press **RETURN** after you type each address.

5. If you make a mistake typing the addresses, don't worry; just press **(RETURN)** to move the blinking cursor to the memory cell with the address you want to change and type the new address over the old. When you are done, press **(ESCAPE)** to return to the Bugbyter command level.

Pressing **(ESCAPE)** always returns you to the Bugbyter command level.

6. To return to Single-Step mode and execute the next instruction, type

**S**

at the command level and press **(RETURN)**.

Bugbyter executes the next instruction of your program, storing the value \$C1 into the memory location \$1100. You can verify that this value actually was stored by looking at the value shown in the Memory subdisplay for that address.

```

C      R      B      PC      A      X      Y      S      P      NV-B013C
0000  00  0      1012  C1  00  C1  FD  B0  10110000

1F7: 43
1F8: 04
1F9: C1
1FA: F1
1FB: 00      1000: LDY  #$C0      P: 10110000
1FC: 01      1002: LDX  #$00      P: 00110010
1FD: 01      1004: JSR  $1000      P: 00110010
1FE: 06      1000: INY              P: 10110000
1FF: 10      100E: TYA              P: 10110000
100: FF      100F: STA  $1100,X    P: 10110000
101: FF      1012: RTS
102: FF      1013: BRK
103: FF      1014: BRK

1100: C1  A      BP      POINT  COUNT  TRIG      BROKE
1101: 00  @      1      0000    0000    0000    0000
1102: 00  @      2      0000    0000    0000    0000
1103: 00  @      3      0000    0000    0000    0000
1104: 00  @      4      0000    0000    0000    0000

```

SINGLE STEP

## Tracing Your Program

To make sure that the rest of your program operates as it should, trace the remainder, rather than single-stepping through it. When Bugbyter traces your program, it updates the Master Display after executing each instruction. You can watch your program fill the memory locations \$1101 through \$1104 with the codes for the characters *B* through *E*.

Pressing **RETURN** from Single-Step mode places you in Trace mode.

1. To trace the remainder of your program, press **RETURN** while you are in Single-Step mode.

You should see the words *SINGLE STEP* on the Bugbyter command line before you press **RETURN**. After you press **RETURN**, Bugbyter replaces this with the word *TRACE* for Trace mode.

Bugbyter traces the rest of your program, executing each instruction in turn. When Bugbyter finishes, it will stop. You can watch the Master Display change as Bugbyter executes your program. When Bugbyter stops, you can verify by looking at the Memory subdisplay that your program correctly loaded the five characters *A* through *E* into memory locations \$1100 through \$1104.

```

C    R    B    PC    A    X    Y    S    P    NV-BDIZC
0000 00 0    100C C5 05 C5 FF 33 00110011

1F9: C1    1003: CPX  #$05    P:10110000
1FA: F1    100A: BNE  $1004    P:10110000
1FB: 00    1004: JSR  $1000    P:10110000
1FC: 01    1000: INY          P:10110000
1FD: 01    100E: TYA          P:10110000
1FE: 06    100F: STA  $1100,X  P:10110000
1FF: 10    1012: RTS          P:10110000
100: FF    1007: INX          P:10110000
101: FF    1008: CPX  #$05    P:00110000
102: FF    100A: BNE  $1004    P:00110011
103: FF    100C: RTS          P:00110011
104: FF    1000: INY          P:00110011
105: FF    100E: TYA

```

```

1100: C1 A  BP  POINT  COUNT  TRIG  BROKE
1101: C2 B  1   0000   0000   0000   0000
1102: C3 C  2   0000   0000   0000   0000
1103: C4 D  3   0000   0000   0000   0000
1104: C5 E  4   0000   0000   0000   0000

```

:\*

To modify a memory location, type an address followed by a colon (:), and then either an assembly-language mnemonic, a numeric constant, or a character string.

---

### Changing Your Program in Memory

Using Bugbyter, you can change your program in memory. This lets you immediately test new versions of your program without having to leave Bugbyter and use the Editor/Assembler to edit and reassemble your source program.

For example, change the first instruction of your program, at address 1000, to LDY #00.

---

1. At the Bugbyter command level, type

```
1000: LDY #00
```

and press **(RETURN)**.

---

Note that you must follow the address you have typed with a colon (:) before you type the assembly-language instruction mnemonic.

---

2. To see how your entire program looks now, type

```
1000L
```

and press **(RETURN)**.

---

Notice that address \$1000 contains the new instruction.

It is also possible to change just one byte in your program. Notice from the Disassembly subdisplay that address \$1009 (the count limit used in controlling the program loop) has the value \$05. To change the number of times that your program runs through this loop, change this limit to \$C0.

---

3. At the Bugbyter command level, type

```
1009: C0
```

and press **(RETURN)**.

---

4. To view your new program, type

```
1000L
```

and press **(RETURN)**.

---

Bugbyter displays your new program in the Disassembly subdisplay.

```

C      R      B      PC      A      X      Y      S      P      NV-BDIZC
0000  00  0      100C  C5  05  C5  FF  33  00110011

1F9:  C1      1000:  LDY  #$00      A0  00
1FA:  F1      1002:  LDX  #$00      A2  00
1FB:  00      1004:  JSR  $1000      20  00  10
1FC:  01      1007:  INX                      E8
1FD:  01      1008:  CPX  #$C0      E0  C0
1FE:  90      100A:  BNE  $1004      00  F8
1FF:  37      100C:  RTS                      60
100:  FF      100D:  INY                      C8
101:  FF      100E:  TYA                      98
102:  FF      100F:  STA  $1100,X      90  00  11
103:  FF      1012:  RTS                      60
104:  FF      1013:  BRK                      00
105:  FF      1014:  BRK                      00
1100: C1  A      8P  POINT  COUNT  TRIC  BROKE
1101: C2  B      1      0000      0000      0000      0000
1102: C3  C      2      0000      0000      0000      0000
1103: C4  D      3      0000      0000      0000      0000
1104: C5  E      4      0000      0000      0000      0000

```

:\*

The **T** command places you in Bugbyter Trace mode and traces your program until it reaches the program end or a breakpoint.

5. To trace this new version of your program, type

1000T

and press **(RETURN)**.

Bugbyter begins tracing your new program starting at location \$1000.

It takes about 30 seconds for Bugbyter to trace your whole program, since you made your program execute the program loop many more times than before. You can watch the X-register count how many times your program has gone through the loop. When Bugbyter finishes executing your program and encounters the RTS instruction at address \$100C, it returns you to the Bugbyter command level.

### Viewing a Page of Memory

The modified version of your program fills nearly a page of memory with the first 192 bytes of the Apple II character set. You can view this area of memory using Bugbyter's Memory Page display.

6. When Bugbyter completes tracing your program, type

1100:  
and press **(RETURN)**.

Bugbyter replaces the Master Display with its Memory Page display: a screenful of memory information starting at location \$1100. Because your program executed the loop \$C0 (192 decimal) times, you see much of the Apple II character set displayed, including inverse-video and blinking characters.

```

1100: 01 02 03 04 05 06 07 08 ABCDEFCH
1108: 09 0A 0B 0C 0D 0E 0F 10 IJKLMNOP
1116: 11 12 13 14 15 16 17 18 QRSTUWXX
1124: 19 1A 1B 1C 1D 1E 1F 20 YZC\J^
1132: 21 22 23 24 25 26 27 28 !"#%&'(
1140: 29 2A 2B 2C 2D 2E 2F 30 )*+,-./0
1148: 31 32 33 34 35 36 37 38 12345678
1156: 39 3A 3B 3C 3D 3E 3F 40 9:;=>?@
1164: 41 42 43 44 45 46 47 48 ABCDEFCH
1172: 49 4A 4B 4C 4D 4E 4F 50 IJKLMNOP
1180: 51 52 53 54 55 56 57 58 QRSTUWXX
1188: 59 5A 5B 5C 5D 5E 5F 60 YZC\J^
1196: 61 62 63 64 65 66 67 68 !"#%&'(
1204: 69 6A 6B 6C 6D 6E 6F 70 )*+,-./0
1212: 71 72 73 74 75 76 77 78 12345678
1220: 79 7A 7B 7C 7D 7E 7F 80 9:;=>?@
1228: 81 82 83 84 85 86 87 88 ABCDEFCH
1236: 89 8A 8B 8C 8D 8E 8F 90 IJKLMNOP
1244: 91 92 93 94 95 96 97 98 QRSTUWXX
1252: 99 9A 9B 9C 9D 9E 9F A0 YZC\J^
1260: A1 A2 A3 A4 A5 A6 A7 A8 !"#%&'(
1268: A9 AA AB AC AD AE AF B0 )*+,-./0
1276: B1 B2 B3 B4 B5 B6 B7 B8 12345678

```

The **Quit** command lets you exit Bugbyter and return to Applesoft BASIC.

7. To return to the Bugbyter command level, press **(ESCAPE)**.  
To exit Bugbyter and return to Applesoft/DOS, type

QUIT  
and press **(RETURN)**.

You are now finished with this tutorial. You learned how to use most of the features of Bugbyter. You know how

- to use the command line;
- to read the Register subdisplay;
- to read the Stack subdisplay;
- to single-step and trace your program;



- to use the Memory subdisplay to view portions of memory;
- to modify your program in memory and test the modified program;
- to use the Memory Page display to view larger portions of memory.

The remainder of this chapter describes in detail each of the debugging functions that you can perform using Bugbyter. The tutorial introduced you to many of these techniques, but the following sections describe these functions more completely and discuss some other Bugbyter features that are not mentioned in the tutorial.

These sections include descriptions of

- starting, relocating, and restarting Bugbyter
- viewing and altering memory or 6502 registers
- altering Bugbyter's Master Display layout
- debugging techniques for controlling the execution of your program, including: single-step mode; trace mode and transparent breakpoints; execution mode and real breakpoints
- debugging real-time code
- debugging programs that use the keyboard and video display.

## Using Bugbyter

Bugbyter is a 6.7K (\$1A00 byte) binary program that must be fully resident in memory whenever you use Bugbyter to test or debug your programs. Since your own program must share the Apple II's memory with Bugbyter, you must be careful that the two programs do not overlap each other in memory.

You load and execute Bugbyter using the DOS BRUN command, just as you did in the preceding tutorial; that is, from either Applesoft or Integer BASIC, type

```
BRUN BUGBYTER
```

and press **RETURN** to run the Bugbyter program. Your Apple II reads the Bugbyter program from your disk and loads it into memory from location \$7C00 to \$95FF. This default starting address (\$7C00) locates Bugbyter just below DOS Buffer One, high enough in memory not to interfere with most small-to-medium-sized assembly-language programs.

A memory map showing all the locations in which your assembly-language program and Bugbyter can reside: see Appendix H.

Unless you specify a starting address with the BRUN command, the Bugbyter program loads starting at memory address \$7C00.

---

### ***Relocating the Bugbyter Program***

If your own program uses memory locations that fall within the block from \$7C00 to \$95FF, you must relocate Bugbyter to some other memory location. You can specify any starting address from \$800 to \$7C00 when you type the BRUN command to execute Bugbyter. For example, to run Bugbyter starting at location \$1000, type

```
BRUN BUGBYTER, A#1000
```

and press **(RETURN)**. Your Apple II loads Bugbyter and executes it from locations \$1000 through \$29FF.

---

### ***Using Bugbyter in a Language Card***

To use Bugbyter in a Language Card or a RAM card, type

```
BRUN BUGLOADER
```

and press **(RETURN)**. The Bugloader program automatically loads and executes Bugbyter in your Language Card or RAM card starting at location \$D000.

When Bugbyter is running in a Language Card, the Language Card must not be write-protected. Bugbyter must be able to store temporary variables within this memory block.

You can also run the Bugbyter from some other location in the Language Card; you do not have to run it at the starting address fixed in the Bugloader. The Bugloader program performs the same function as if you type from either BASIC:

```
CALL -151  
0081 0081 F800<F800, FFFFM 0083 0083  
BRUN BUGBYTER, A$0000
```

These three lines

1. call the Apple II Monitor;
2. copy the Monitor from ROM into the Language Card;
3. load and execute Bugbyter at the first location in the Language Card.

To execute Bugbyter at another location within the Language Card, you can retype these three lines yourself and BRUN Bugbyter at any address from \$D000 to \$DE00.

The Bugbyter program is self-modifying; that is, once you BRUN the program at a certain address, Bugbyter modifies itself so it can execute at that address. For this reason, if you want to move Bugbyter around in memory you should BLOAD Bugbyter before moving it, rather than using BRUN.

For example, if the program you are debugging does not require DOS, you might want to run Bugbyter at address \$A600 to leave more room for your own program. To do this, BLOAD Bugbyter at some lower address (\$7C00, for example), and then use the Monitor to move Bugbyter to start at address \$A600.

---

### Entering the Monitor

From the Bugbyter command level, you can enter the Apple II Monitor to use the Monitor's block memory moves and other capabilities not provided by Bugbyter. Your *Apple II Reference Manual* describes the features of the Apple II Monitor.

To enter the Monitor, type

M

and press **RETURN**. After you finish using the Monitor, type

**CONTROL**-Y

and press **RETURN** to return to the Bugbyter command level.

---

### Restarting Bugbyter

If you should ever exit Bugbyter for any reason, either into the Monitor or into BASIC, you can use the Monitor's **CONTROL**-Y vector to restart Bugbyter, or you can use Bugbyter's load address.

For example, if you run Bugbyter by typing

BRUN BUGBYTER, A\$2000

and press **RETURN**, you can restart Bugbyter from either Applesoft or Integer BASIC by pressing **RESET** and typing either

CALL 1016      This uses the Monitor's **CONTROL**-Y vector.

or

CALL 8192      This uses Bugbyter's specific load address.

and pressing **RETURN**.

From the Monitor, you can reenter Bugbyter by pressing

**CONTROL-Y**

and then **RETURN**.

---

### ***Memory and the Bugbyter Displays***

In the Bugbyter tutorial, you were introduced to many of Bugbyter's features for viewing and altering the contents of memory locations and registers. The following section describes in more detail how you can do this and also describes how you can alter the layout of Bugbyter's Master Display.

Using Bugbyter, you can view the contents of memory in several different ways.

- You can use the Stack and Disassembly subdisplays of the Master Display to show the contents of specific regions of memory. Select the region that shows the Disassembly subdisplay by using the Load (L) command described in the tutorial.
- You can use the Memory subdisplay of the Master Display to show the contents of several individual memory locations. Use the MEM command to set particular memory addresses for each cell of this subdisplay.
- You can use the Memory Page display to show the contents of 184 (\$B8) contiguous memory locations both as hexadecimal values and as Apple II characters.

---

#### ***Using the Memory Subdisplay***

The Memory Cell subdisplay of Master Display continuously displays the contents of several individual memory locations that you can select. You can use the MEM command to set the memory addresses in one or more cells of this display. If you want, you can first use the Set command to increase the number of cells in this subdisplay.

The **MEM** command lets you modify Bugbyter's Memory subdisplay.

To set addresses in the Memory subdisplay from the Bugbyter command level, type

MEM

and press **(RETURN)**. The cursor moves to the first address at the top of the Memory subdisplay. You can now choose

- to type a smaller than four-digit hexadecimal address for this memory cell, and press **(RETURN)**;
- to type a four-digit address; the cursor automatically moves to the next address;
- to use **(←)**, **(→)**, or **(RETURN)** to move the blinking cursor to another memory cell.

Unless you specify otherwise, Bugbyter displays the contents of each memory cell in hexadecimal notation and as a character.

Before typing an address into a memory cell, you can select whether Bugbyter will display the contents of that memory location as both a hexadecimal value and an Apple II character, or as a four-digit hexadecimal address. Before typing the address into a memory cell, type

H,

to display the contents of the location in hexadecimal and as a character, or

P

to display the contents of the two consecutive locations (address+1 and address) as a four-digit hexadecimal address pointer (most significant byte first).

To return to the Bugbyter command level, press **(ESCAPE)**.

---

### Viewing the Memory Page Display

To view an entire screenfull of memory locations (two-thirds of a 6502 memory page) from the Bugbyter command level, type the starting address of the memory block, type a colon (:), and press **(RETURN)**. For example, if you just completed the tutorial and have the Apple II character set in memory at address \$1100, type

1100:

and press **(RETURN)** to tell Bugbyter to replace the Master Display with the Memory Page display having the address 1100 in the upper-left corner.

Press **(SPACE)** and then **(RETURN)** to display the next memory page.

```

1100: 01 02 03 04 05 06 07 08 ABCDEFGH
1108: 09 0A 0B 0C 0D 0E 0F 10 IJKLMNOP
1110: 11 12 13 14 15 16 17 18 QRSTUVWX
1118: 19 1A 1B 1C 1D 1E 1F 20 YZC\J
1120: 21 22 23 24 25 26 27 28 !"#$%& (
1128: 29 2A 2B 2C 2D 2E 2F 30 )*+,-./0
1130: 31 32 33 34 35 36 37 38 12345678
1138: 39 3A 3B 3C 3D 3E 3F 40 9:;<=>?@
1140: 41 42 43 44 45 46 47 48 ABCDEFGH
1148: 49 4A 4B 4C 4D 4E 4F 50 IJKLMNOP
1150: 51 52 53 54 55 56 57 58 QRSTUVWX
1158: 59 5A 5B 5C 5D 5E 5F 60 YZC\J
1160: 61 62 63 64 65 66 67 68 !"#$%& (
1168: 69 6A 6B 6C 6D 6E 6F 70 )*+,-./0
1170: 71 72 73 74 75 76 77 78 12345678
1178: 79 7A 7B 7C 7D 7E 7F 80 9:;<=>?@
1180: 81 82 83 84 85 86 87 88 ABCDEFGH
1188: 89 8A 8B 8C 8D 8E 8F 90 IJKLMNOP
1190: 91 92 93 94 95 96 97 98 QRSTUVWX
1198: 99 9A 9B 9C 9D 9E 9F A0 YZC\J
11A0: A1 A2 A3 A4 A5 A6 A7 A8 !"#$%& (
11A8: A9 AA AB AC AD AE AF B0 )*+,-./0
11B0: B1 B2 B3 B4 B5 B6 B7 B8 12345678
:~

```

Your display might show different values, depending on the current contents of this region of memory.

Bugbyter displays the contents of memory in a table with eight memory locations to a line. The contents of each memory location are shown first in hexadecimal and then as the equivalent Apple II character. Each line of this table starts with the memory address of the first byte shown on that line. The normal Apple II character set is:

00-3F inverse-video characters  
40-7F flashing-video characters  
80-FF normal-video characters (two sets of alphabetic characters)

Bugbyter displays the colon prompt on the last line of this display and will accept either another address followed by a colon to display another memory page, or a memory assignment command.

The memory assignment command: see the section "Altering the Contents of Memory."

To return to the Bugbyter command level and the Master Display, press ESCAPE.



---

### ***Altering the Contents of Memory***

Using Bugbyter you can change the contents of any RAM memory location in your Apple II. From the Bugbyter command level or from the Memory Page display, you can change one or a sequence of memory locations by typing a hexadecimal address followed by a colon and

- one or more hexadecimal bytes
- one or more character strings (enclosed in double or single quotes)
- combinations of the two above
- 6502 instruction mnemonics and operands.

You can freely mix hexadecimal numbers and character strings of any length in your memory assignment commands, separating each item with one or more spaces.

When you are entering character strings, Bugbyter stores the characters with their most significant bit *on* if you enclose the characters in double quotes (""). Bugbyter stores these characters with their most significant bit *off* if you enclose the character string in single quotes.

For example, if you type

```
$805: "HELLO" 8D
```

and press **(RETURN)**, Bugbyter fills the memory locations from \$805 to \$80A with the bytes \$C8, \$C5, \$CC, \$CC, \$C5 ("HELLO"), followed by a byte with the value of \$8D. Since you used double quotes to delimit the character string, the character codes are stored with their most significant bit on.

Bugbyter also accepts 6502 instruction mnemonics in a memory assignment command. You must use the standard address mode syntax when specifying operands, and you must type each assembly-language instruction in a separate memory assignment command. For example, if you type

```
1000: LDY #$80
```

and press **(RETURN)**, Bugbyter assembles this statement and stores the two resulting bytes (\$A0 and \$80) in the memory locations \$1000 and \$1001.



Any time you alter a memory location that is currently displayed on either the Master Display or the Memory Page display, Bugbyter immediately updates the display to show any changes.

---

### ***Altering the Contents of Registers***

The Bugbyter Master Display always shows the current contents of the 6502 registers as they appear to the program you are debugging. You can change the contents of these registers at any time from the Bugbyter command level.

To change the contents of a 6502 register from the Bugbyter command level, type the name of the register followed by an equal sign (=), the value to be stored in the register, and press **RETURN**. For example, if you type

A=\$D

and press **RETURN**, Bugbyter sets the contents of the A-register to \$D. This value is immediately reflected under the A-register label in the Register subdisplay. In the same manner, you can change the C, R, PC, A, X, Y, S, and P registers to any value you wish. The Bugbyter's B flag is not a register, and you cannot change this flag using this register-assignment command. The NV-BDIZC display is the binary representation of the P-register. To change individual bits in the P-register, type F= followed by the appropriate hexadecimal number.



---

### **Warning**

Bugbyter uses the first 32 bytes of the 6502's stack (locations \$100 to \$11F), and any attempt by you or your program to alter the stack pointer to point into this region may result in a collision between Bugbyter and the program you are debugging.

Bugbyter flashes a warning in the Stack subdisplay area if you or your program sets the stack pointer to any value less than \$20 (Since the stack pointer points only to locations in page one of memory, a stack pointer value of \$20 implies a memory address of \$120).

---

**By the Way:** To help you when you are assigning numeric values to memory locations or registers, or whenever you are converting numbers from one base to another, the Bugbyter performs simple conversions from hexadecimal to decimal, or from decimal to hexadecimal. For example, from the Bugbyter command level, if you type

`$C3=`

and press **RETURN**, Bugbyter converts the hexadecimal number `$C3` to its decimal equivalent and display this value following the equal sign:

`$C3=00195`

In an equivalent manner, you can type hexadecimal numbers without the dollar sign (\$). For example, if you type

`C3=`

and press **RETURN**, Bugbyter treats this command the same way it did the `$C3=` command above.

To convert a decimal number to its hexadecimal equivalent, you must precede the decimal number with a plus (+) or minus (-) sign to distinguish it from a hexadecimal number. For example, if you type

`+43=`

and press **RETURN**, Bugbyter responds by displaying the hexadecimal equivalent following the equal sign on the command line:

`+43=$2B`

---

### ***Altering Bugbyter's Master Display Layout***

Bugbyter's Master Display has several subdisplays that let you view many different items of information at once. You can customize Bugbyter's Master Display layout to alter the relative sizes of some of these subdisplays, allowing you

- to set more or fewer program breakpoints—the Breakpoint subdisplay;
- to view larger or smaller portions of your disassembled program—the Code Disassembly subdisplay;
- to view larger or smaller regions of the memory stack—the Stack subdisplay;
- to view more or fewer memory cells—the Memory subdisplay.

The **SET** command lets you set the relative sizes of the various Bugbyter subdisplays.

To change the relative sizes of these subdisplays, you use the Set command. From the Bugbyter command level, type

**SET**

and press **(RETURN)**. Bugbyter displays a sketch of the Code Disassembly and the Breakpoint subdisplays and allows you to set the relative sizes of these displays.

Now you can

- use **(←)** to increase the number of breakpoints that are displayed and simultaneously decrease the size of the Code Disassembly subdisplay;
- use **(→)** to decrease the number of breakpoints that are displayed and simultaneously increase the size of the Code Disassembly subdisplay.

When you are satisfied with the relative sizes of these subdisplays, press **(RETURN)** to fix the relative sizes as you have selected them.

You can now select where Bugbyter displays the "next instruction to be executed" bar in the Code Disassembly subdisplay. Use the **(←)** and **(→)** to adjust the position of the inverse-video bar within the Code Disassembly display. The position of this bar divides the subdisplay into the instructions that have been executed (above the bar) and instructions not yet executed (at and below the bar) when you are tracing or single-stepping your program. Press **(RETURN)** when you are satisfied with the bar position.

Bugbyter then displays a sketch of the Stack and Memory Cell subdisplays. Use the **(←)** and **(→)** to adjust the relative sizes of the Stack and Memory subdisplays, just as you did for the Disassembly and Breakpoint subdisplays.

When you are satisfied with the relative sizes of these displays, press **(RETURN)** to fix the sizes. Then use the **(←)** and **(→)** to adjust the position of the stack-pointer bar within the Stack subdisplay. Typically, you want this bar near the top of the subdisplay, since information on the stack appears below this inverse-video bar.

When you are done, press **(RETURN)** to return to the Bugbyter command level.

When you use the Set command, you do not change the contents of the Memory subdisplay or the Breakpoint subdisplay. You can recover any Memory cells or breakpoints that are no longer displayed by using the Set command to enlarge these subdisplays and to display this information once again.

---

### ***Controlling the Execution of Your Program***

There are probably as many ways to debug a program as there are ways to write a program; both are related to a programmer's particular style. This manual is not intended to teach you a particular way to debug your programs any more than it is intended to teach you programming style. There are, however, several techniques for controlling the execution of your program that are very useful for debugging. This section discusses some debugging techniques that you can use with Bugbyter; they include

- single-stepping your program
- tracing your program
- executing your program directly on the 6502.

---

### ***Using the Single-Step and Trace Modes***

Bugbyter's Single-Step and Trace modes provide you with a powerful debugging environment. Bugbyter is capable of tracing almost any executable 6502 program, including interrupt and timing-sensitive code. You have already found the Single-Step and Trace modes easy to use if you followed the Bugbyter tutorial.

When you enter Single-Step or Trace mode, Bugbyter removes the command prompt from the command line and replaces it with the words SINGLE STEP or TRACE. Once you enter one of these modes, you can no longer type commands at the Bugbyter command level. Instead, you can type a set of single-keystroke commands that give you access to a variety of debugging features. To return to the Bugbyter command level, simply press **[ESCAPE]**.

Table 4-2 shows the functions that you can perform and the single keystrokes that you use.

**Table 4-2.** Trace and Single-Step  
Keystroke Commands

Trace or Single-Step Operation	Keystroke Command
Return to Bugbyter command level	<b>ESCAPE</b>
Single step (execute) one instruction	<b>SPACE</b>
Skip next instruction	<b>-</b>
Trace program until a breakpoint or end instruction	<b>RETURN</b>
Trace until an RTS	<b>R</b>
Clear Cycle Count register	<b>C</b>
Use hand control 0 to adjust Trace Rate	<b>P</b>
Use Keyboard Rate to adjust Trace Rate; (set R=value before entering Trace mode)	<b>K</b>
Turn off Bugbyter Sound (Quiet)	<b>Q</b>
Turn on Bugbyter Sound	<b>S</b>
Display primary screen	<b>1</b>
Display secondary screen	<b>2</b>
Display Text screen	<b>T</b>
Display Low-resolution graphics screen	<b>L</b>
Display High-resolution graphics screen	<b>H</b>
Display Full screen graphics	<b>F</b>
Display Mixed graphics screen, with Bugbyter command line visible	<b>M</b>

To access any of these debugging functions, you need type only the single keystroke. To return to the Bugbyter command level and view the Master Display, press **ESCAPE**.

You can reenter Single-Step mode from the Bugbyter command level to continue single-stepping by typing

**S**

and pressing **RETURN**, or you can enter Trace mode by typing

**T**

and pressing **RETURN**. Bugbyter remembers the last instruction it executed before leaving Trace or Single-Step mode, and continues from the next instruction whenever you reenter Trace or Single-Step mode without specifying a starting address.

### ***Single-Stepping Your Program***

When you debug a program, you spend much of your time observing the execution of your program and verifying that it works correctly. The single-step function of the Bugbyter was introduced in the Bugbyter tutorial. In Single-Step mode, Bugbyter allows your Apple II to execute a single instruction at a time, stopping after each instruction to allow you to observe the result.

To enter Single-Step mode from the Bugbyter command level, type the memory address of the first instruction that you wish to execute, followed by an (S), and press (RETURN). In the example in the tutorial, you typed

10000

After you type the single-step command, Bugbyter shows in the Disassembly subdisplay the first instruction to be executed. To execute this instruction, press (SPACE). Bugbyter executes this single instruction and updates the screen to show you any effects that occurred due to the execution of the instruction.

To continue executing succeeding instructions, press (SPACE) for each instruction to be executed. After Bugbyter executes each instruction, it updates its Master Display, allowing you to verify that your program has executed correctly, or showing you exactly where your program went awry.

### ***Using Trace Mode***

You will find Bugbyter's Single-Step mode very useful for observing short programs or small portions of larger programs. For completely testing programs that are medium to large in size, however, single-stepping through each instruction is rather slow and tedious.

To test portions of medium to large programs, you will find it more efficient to let Bugbyter execute whole portions of your program at a time, interrupting this execution at places you specify to allow you to observe program results and the status of registers and the stack. This technique lets you skip over portions of your program that are operating correctly and allows you to get swiftly to the location of a problem.



The **R** command allows you to trace your program up to the occurrence of the next RTS instruction. This provides an easy way to debug individual subroutines within your program.

### **Tracing Subroutines**

Most good programs are written using subroutines as a basic building block. Rather than single-step your program one instruction at a time, Bugbyter allows you to use Trace mode to step easily through your program one subroutine at a time.

You can activate this feature only from Bugbyter's Single-Step mode. To trace your program until the next occurrence of an RTS instruction, that is, until the end of the next subroutine, type

**R**

while you are in Single-Step mode. Bugbyter begins executing each instruction of your program in sequence, updating the Master Display after each instruction, until it encounters an RTS instruction in your program or a breakpoint. When Bugbyter encounters an RTS instruction, it returns you to Single-Step mode and lets you observe the results of the subroutine and continue debugging as you wish.

### **Setting Transparent Breakpoints**

A common way to control Bugbyter's trace of your program is to set breakpoints at particular locations within your program.

When you are debugging in Trace or Single-Step mode, Bugbyter monitors your Apple II's program counter (PC) before executing each instruction and compares it to any breakpoints that you may have set in the Breakpoint subdisplay. If your program reaches one of these breakpoints (that is, if the 6502's program counter matches the address of a breakpoint), Bugbyter interrupts your program and returns you to Bugbyter's command level.

A **transparent breakpoint** does not alter your program's code in any way. A **real breakpoint**, used in Execution mode, actually alters your program temporarily.

If you do not set any breakpoints, or if Bugbyter does not encounter any while executing your program, Bugbyter continues executing your program until you press **(ESCAPE)**, or until it reaches the end of your program.

To set transparent breakpoints, you should be familiar with the breakpoint subdisplay area (Figure 4-8) of Bugbyter's Master Display.



Figure 4-8. The Breakpoint Subdisplay

C	P	B	PC	A	C	V	S	P	NW-B0100
0000	00	0	0000	00	00	00	FF	02	00000010
1F3:	01								
1F4:	01								
1F5:	00								
1F6:	01								
1F7:	01								
1F8:	00								
1F9:	01								
1FA:	00								
1FB:	00								
1FC:	00								
1FD:	00								
1FE:	00								
1FF:	00								
100:	00								
101:	00								
102:	00								
103:	00								
104:	00								
105:	00								
0000:40	L								
0000:40	L								
0000:40	L								
0000:40	L								
0000:40	L								

BP	POINT	COUNT	TRIG	BROKE
1	0000	0000	0000	0000
2	0000	0000	0000	0000
3	0000	0000	0000	0000
4	0000	0000	0000	0000

BP	POINT	COUNT	TRIG	BROKE
1	0000	0000	0000	0000
2	0000	0000	0000	0000
3	0000	0000	0000	0000
4	0000	0000	0000	0000

The Breakpoint subdisplay is in the lower-right section of Bugbyter's Master Display. You can use the Set command to increase or decrease the number of breakpoints that are displayed. The Breakpoint subdisplay has four column headings, where each line under these four headings represents one breakpoint. For each breakpoint

- POINT is the hexadecimal address of the breakpoint.
- COUNT is the number of times that Bugbyter has encountered this breakpoint address while executing your program since the Bugbyter last Triggered at this breakpoint.
- TRIG is a count that you can specify. Bugbyter does not interrupt program execution until it encounters this particular breakpoint the number of times that you specify. A TRIG count of 1 causes Bugbyter to interrupt program execution the first time it encounters this breakpoint.
- BROKE is the number of times Bugbyter has actually Triggered, or interrupted program execution, at this particular breakpoint.

To enter a breakpoint address from the Bugbyter command level, type EF followed by a breakpoint number. For example, to set a breakpoint to be displayed in row number 1 of the Breakpoint subdisplay, type

EF1

and press **(RETURN)**.

To set a breakpoint address, type EF followed by a breakpoint number.

When you type this breakpoint setup command, Bugbyter moves the cursor to the first zero in the POINT field of the breakpoint row you specify. Enter a hexadecimal address in this field to represent the address of breakpoint #1. Use **(←)** to move the cursor to the TRIG field on that same breakpoint line. In the TRIG field, type a

hexadecimal number greater than zero. If you want Bugbyter to stop on the first occurrence of this breakpoint, type 1 in this field. If you leave TRIG set to 0, Bugbyter ignores this breakpoint. Press **(RETURN)** when you finish setting the breakpoint address and the TRIG count.

You typically might set breakpoints at critical locations in your program, such as just after a call to an important subroutine. When you are tracing your program, you can then verify that the results from this subroutine are correct and are stored in the proper registers or memory locations. Other locations to test might be after an important compare instruction, or at any place you want to check the status of your program.

#### **Using Breakpoints**

Once you set breakpoints where you want them, you can use Trace or Single-Step mode to monitor the execution of your program. Every time Bugbyter encounters an instruction located at a breakpoint address (Point), Bugbyter increments the Count for that breakpoint and compares this Count to the Trig value you set for that breakpoint. When the Count equals your Trig value, Bugbyter stops your program execution *before* executing the instruction at the Point address. Bugbyter then highlights the row in the Breakpoint subdisplay corresponding to the breakpoint that triggered and clears the Count.

You can then observe all the 6502 registers, stack, and other conditions that existed just before your program was interrupted by the occurrence of the breakpoint. If you want to continue executing your program from the point where the breakpoint occurred, type

T

and press **(RETURN)** to reenter Trace Mode, or type

S

and press **(RETURN)** to enter Single-Step mode.

#### **Clearing Transparent Breakpoints**

To clear a particular breakpoint, type CLR followed by the breakpoint number, and press **(RETURN)**.

The **CLR** command allows you to clear one or all transparent breakpoints.

To clear all breakpoints, type

CLR

and press **(RETURN)**.

### **Adjusting the Trace Rate**

During tracing, Bugbyter interprets each 6502 instruction of your program. This means that the 6502 microprocessor is executing the Bugbyter program, which in turn is executing your program code. The result of this process is that the code you are tracing using Bugbyter executes much slower than if it were executed directly by the 6502 microprocessor. You can adjust the rate at which Bugbyter traces your code by one of several methods.

- Before you enter Trace mode, set a value in Bugbyter's Trace Rate register, displayed in the Register subdisplay. Type `R=`, followed by a hexadecimal value from 0 to FF, where 0 is the fastest rate and FF is the slowest. Press `(RETURN)`. When you enter Trace mode, Bugbyter uses this rate setting to control the speed of its trace.
- If you have hand controls for your Apple II, use them to adjust the trace rate while you are in Trace mode. While in Trace mode, press `(P)` and use hand control 0 to adjust the trace rate. To return to the keyboard (R-register) rate, press `(K)` to disable the hand control.
- Before you enter Trace mode, type `OFF` from the Bugbyter command level and press `(RETURN)`. This clears Bugbyter's Master Display. Turning off the Master Display greatly increases the speed of tracing once you enter Trace mode. To restore the Master Display, type `ON` and press `(RETURN)` from the Bugbyter command level, or press `(ESCAPE)`, type `ON`, and then press `(RETURN)` from Trace mode.

### **Using Display Options in Trace and Single-Step Mode**

You can select one of seven display options for Bugbyter to use during Trace and Single-Step mode to display different types of relevant information.

To select one of these options, you must do so from the Bugbyter command level before you enter Trace or Single-Step mode. Table 4-3 shows the display options that you can select and the commands you can use to select a particular option. You can select only one option at a time; Bugbyter displays only the most recently selected option.

The Bugbyter display options are shown at the right edge of the Code Disassembly subdisplay, just to the right of each associated instruction.

**Table 4-3.** Code Disassembly Display Options

Display Option	To Select, type command and press <b>RETURN</b> :
A-register in binary	O=A
X-register in binary	O=X
Y-register in binary	O=Y
Stack Pointer in binary	O=S
Processor Status Byte in binary	O=P
Machine-instruction bytes in hex	O=B
Computed effective addresses, relative branches, and instruction cycles	O=E

The last option shown in Table 4-3, the O=E display option, requires some extra discussion. There are four 6502 addressing modes for which the 6502 internally computes an effective address. These modes are:

Mode	Example
indexed	LDA \$300,X
indirect	JMP (\$300)
indexed indirect	LDA (\$10,X)
indirect indexed	LDA (\$10),Y

During Trace or Single-Step mode, Bugbyter computes the actual or effective address for each instruction using the current contents of registers or memory cells at the time the instruction is executed. If you select the O=E option, Bugbyter displays these effective addresses in the Disassembly subdisplay. Bugbyter also displays all relative branch offsets (the hex byte operand of a branch opcode) when you select this option.

When you select the O=E option, Bugbyter displays the number of instruction cycles used by each instruction. This count of the number of cycles appears in parentheses to the right of the Disassembly line for each instruction that is executed.

As an example of this display option, if you type a command followed by (RETURN), the command

---

C=0	clears the Cycle Count register in the Register subdisplay
O=E	sets the Display Option to E
A=12	sets the accumulator to \$12
FCAS6	activates Single-Step mode at the start of the Monitor WAIT routine
R	starts Bugbyter tracing until an RTS instruction is encountered

---

Bugbyter begins tracing the Monitor WAIT routine; the command line at the bottom of the Master Display shows

TRACE Awaiting RTS

The rest of the Master Display changes rapidly. You can watch the Cycle Count register shown in the upper-left corner of the Master Display incrementing after each instruction is executed. After a few seconds, the Master Display halts and shows

```

C 041E R 00 B 00 PC FC83 A 00 X 00 Y 00 S FF P 33 NV-BDIZC
1F9: 7C FCAC: BNE $FCBA B: FC (2)
1FA: 7C FCAD: PLA B: (4)
1FB: A1 FCAD: SEC #$01 B: (2)
1FC: 02 FC81: BNE $FCA9 B: F6 (3)
1FD: E3 FCA9: PHA B: (3)
1FE: 06 FCA9: SEC #$01 B: (2)
1FF: 01 FCAC: BNE $FCBA B: FC (2)
100: FF FCAD: PLA B: (4)
101: FF FCAD: SEC #$01 B: (2)
102: 00 FC81: BNE $FCA9 B: F6 (2)
103: 00 FC83: RTS
104: FF FC84: INC $42
105: FF FC86: BNE $FCBA

0000:4C L 6P POINT COUNT TRIC BROKE
0000:4C L 1 0000 0000 0000 0000
0000:4C L 2 0000 0000 0000 0000
0000:4C L 3 0000 0000 0000 0000
0000:4C L 4 0000 0000 0000 0000

```

SINGLE STEP

The Cycle Count register in the upper left-corner shows \$41E (1054 decimal) processor cycles, representing the time required to execute the WAIT routine when the accumulator is preset to \$12. Using a cycle time of one microsecond, this cycle count represents a WAIT of approximately one millisecond (0.001 seconds).

Bugbyter increments the Cycle Count register only when you select the O=E display option before entering Trace or Single-Step mode. The Cycle Count register is also not incremented if you turn off the Bugbyter Master Display using the Off command.

---

### **Using Execution Mode**

There may be times when you want to execute portions of your program at the full speed of the 6502 microprocessor, without having Bugbyter continuously check for breakpoints and so slow down your program's execution. You can do just this by using Bugbyter's Execution mode.

In Execution mode Bugbyter executes your program directly, at the full speed of the 6502 microprocessor. To debug your program in Execution mode, you must use real breakpoints, rather than transparent breakpoints, to control your program's execution.

A real breakpoint is a 6502 Break opcode (\$00) that Bugbyter places at the breakpoint address in your program code. Real breakpoints differ from transparent breakpoints in that Bugbyter must alter your program code when it inserts real breakpoints; transparent breakpoints do not change your program code in any way.

### **Setting Real Breakpoints**

The **IN** command causes Bugbyter to insert 6502 BRK opcodes at all breakpoint addresses shown in the Breakpoint subdisplay.

To set real breakpoints at the addresses shown in the Point column of the Breakpoint subdisplay, type

IN

from the Bugbyter command level, and press **RETURN**. This command causes Bugbyter to insert 6502 BRK opcodes (00) at all the breakpoint addresses (all Point addresses with their associated Trig's set to greater than zero). Bugbyter displays an **I** for *breakpoints IN* under the B-flag in the register subdisplay at the top of the screen.

Once you insert real breakpoints in your program code, you can debug your program while executing it at the full speed of the 6502 microprocessor.



The **OUT** command causes Bugbyter to remove all the real breakpoints inserted by a previous **In** command.

When real breakpoints are in your program, Bugbyter does not allow you to add, change, or clear any breakpoints. To modify any of the breakpoint information, you must first instruct Bugbyter to take real breakpoints out. Only after you do this will Bugbyter allow you to modify your breakpoints.

To take the real breakpoints out of your program code, type

**OUT**

from the Bugbyter command level and press **(RETURN)**. This command forces Bugbyter to remove the break opcodes that it inserted when you last used the **In** command, and restores the 6502 instruction opcodes originally stored there. The B-flag that monitors the status of real breakpoints in the Register subdisplay at the top of the screen displays the letter **O** for *breakpoints Out*.



#### **Warning**

Once you set real breakpoints in your program, Bugbyter alters your program by inserting break opcodes at every breakpoint address. If you exit Bugbyter before you remove the real breakpoints, your code may be riddled with unwanted 6502 breaks. Be sure to type

**OUT**

and press **(RETURN)** to return your code to its original condition.

If you enter Trace mode when real breakpoints are in, Bugbyter operates just as it does when real breakpoints are out. You will find it more convenient to use transparent breakpoints if you wish to debug your program in Trace or Single-Step mode; real breakpoints are only necessary when you are debugging your program in Execution mode.

#### **Debugging Your Program in Execution Mode**

To execute your program directly from the Bugbyter command level, type a starting address followed by **G** (for **GO**) and press **(RETURN)**, or simply type **G** and press **(RETURN)**.

When you type this command, Bugbyter enters Execution Mode and starts executing your program from the starting address that you indicated. If you didn't type a starting address, Bugbyter uses the last starting address that you specified with either the Single-Step, Trace, or Go commands. For example, to execute your program starting at location \$1000, type

1000G

The **G** command causes Bugbyter to enter execution mode and begin executing your program directly.



The **J command** allows you to resume execution of your program after execution was halted by a breakpoint.

and press **(RETURN)**. Bugbyter treats this command similarly to the Monitor G command: Bugbyter pushes a return address onto the stack before executing your program code. If your program encounters a return-from-subroutine (RTS) instruction, you return to the Bugbyter command level. The same result occurs if your program encounters a BRK opcode or a breakpoint.

To continue executing your program after a breakpoint has forced Bugbyter out of Execution mode, type

J

and press **(RETURN)**. This command does not push a return address onto the stack, assuming that you have already set up Execution mode using the G command. When first executing your code, type

<starting-address>G

and press **(RETURN)**. After encountering any break opcodes, type

J

and press **(RETURN)** to continue direct real-time execution.

**Note:** Because the J command does not push a return address onto the stack, you should always begin debugging in Execution Mode using the G command. If you start Execution mode with the J command and your program encounters an RTS instruction, the 6502 uses an undefined address from the stack with unpredictable results.

When the 6502 encounters a BRK opcode, that is, either a breakpoint inserted by Bugbyter or part of your program code itself, the 6502 passes control back to the Bugbyter program. If the BRK opcode was a breakpoint inserted by Bugbyter, the breakpoint is shown in inverse-video in the Breakpoint subdisplay, and you return to the Bugbyter command level. If the BRK opcode was not a breakpoint inserted by Bugbyter, control passes to the Monitor.

---

### ***Debugging Real-Time Code***

If you are debugging a program that contains real-time sensitive code or calls real-time Apple II DOS routines, these portions of your program will not function correctly if you try to trace their execution in Bugbyter Trace or Single-Step mode.

A **soft switch** is a memory location near the beginning of the Bugbyter program that allows you to control some of Bugbyter's features. A complete list of the Bugbyter soft switches: see Appendix A.

**Table 4-4.** Bugbyter Real-Time Soft Switches

There are two ways that you can debug this type of program using Bugbyter, one of which has been described in the section "Using Execution Mode." The other alternative is to use the more powerful debugging capabilities of Bugbyter's Trace or Single-Step modes to debug the timing-insensitive portions of your program, while still executing the real-time routines of your program at the full speed of the 6502 microprocessor.

Prime examples of real-time sensitive code are the core routines associated with the Apple II DOS operating system. The read-data, write-data, read-address and track-seek routines are very sensitive to cycle speed variation. These core routines will not function at all if you trace them using Trace or Single-Step mode. To execute these subroutines from a program that you are tracing using Bugbyter, you must use Execution mode.

Bugbyter allows you to execute subroutines in Execution mode while tracing your calling routines in Trace or Single-Step mode. You can set up this method of operation by designating a region of memory containing the real-time routines that Bugbyter always executes in Execution mode. You do this by setting two **soft-switch** addresses within Bugbyter.

Offset from the beginning of the Bugbyter program, at locations  $\text{start} + \$0A$  and  $\text{start} + \$0B$ , are the two bytes that you can set to define the starting address of this real-time region. At locations  $\text{start} + \$0C$  and  $\text{start} + \$0D$  are two bytes that indicate the ending address of this region. Table 4-4 lists these real-time soft switches.

Soft-Switch Byte	Relative Address	Absolute Address
High-byte of region start address	$\text{start} + \$0A$	7C0A
Low-byte of region start address	$\text{start} + \$0B$	7C0B
High-byte of region ending address	$\text{start} + \$0C$	7C0C
Low-byte of region ending address	$\text{start} + \$0D$	7C0D

When you are tracing your program in Trace or Single-Step mode, any subroutine (JSR) calls to locations inside this defined region cause Bugbyter to transfer control of your program from Trace or Single-Step mode to Execution mode. This means that the subroutines in this region execute at the full speed of the 6502 microprocessor. When the 6502 encounters the return from subroutine (RTS) instruction that transfers control back to your calling routine, Bugbyter reactivates Trace or Single-Step mode and continues executing your main or calling program.

For example, if you have a 48K version of DOS in memory, with Bugbyter loaded at \$7C00, type

```
7C00: 0 B8 FF FF
```

and press **(RETURN)** to designate a real-time region extending from location \$B800 to \$FFFF; this includes the DOS core routines, RWTS, peripheral I/O, BASIC, and the Monitor). If you then type

```
300: JSR A56E
```

which is the starting address of the DOS Catalog routine, and type

```
303: BRK
```

you set up a calling program for the DOS Catalog routine. Then type

```
OFF
```

to turn the Master Display off, and type

```
300T
```

to trace your calling program.

This command sequence causes Bugbyter to begin tracing DOS's Catalog routine until it encounters a JSR to the Read-Write-Track-Sector (RWTS) routine at \$BD00—inside your real-time region (\$B800-FFFF). At this time, Bugbyter enters Execution mode and allows RWTS to execute directly under the 6502 microprocessor—seeking tracks and reading sectors from the disk in real-time. When the RWTS routine exits to DOS (RTS), Bugbyter reenters Trace mode and continues tracing the code that follows the call to RWTS (JSR A56E).

---

### ***Debugging Programs that Use the Keyboard and Display***

Bugbyter is a screen-based debugger and so requires the use of the screen in order to relay information to you. Likewise, many of Bugbyter's features require you to type commands at the keyboard while you are debugging. To debug your own programs that use the keyboard and screen, you must take care to eliminate contention between Bugbyter and your own program for the use of the keyboard and display.

The **Off** command clears the first 23 lines of the text page, allowing your program to use your video screen without interference from Bugbyter.

The **On** command returns Bugbyter to its normal full-screen display mode following a previous Off command.

A soft switch controls Bugbyter's polling of the keyboard, allowing you to debug programs that require extensive keyboard input.

### ***Eliminating Screen Contention***

Since Bugbyter displays all information on the screen using absolute screen addressing, any programmed output by your program using the Apple II's I/O hooks (CSW) is not impeded. However, since Bugbyter is constantly updating the Master Display when you are in Trace or Single-Step mode, you are not likely to see any of your program's output.

Eliminating this contention between Bugbyter and your program for the use of the Apple II's text screen is a simple matter. From the Bugbyter command level, just type

**OFF**

and press **(RETURN)**. Bugbyter clears the first 23 lines of the text page, leaving the command prompt appearing on the lowest line. When you enter Trace or Single-Step mode, Bugbyter allows your program to write anything it desires to this text page.

To recall the Master display from the Bugbyter command level, simply type

**ON**

and press **(RETURN)** in response to the Bugbyter command prompt. If you are currently in Trace or Single-Step mode, press **(ESCAPE)** first to exit to the Bugbyter command level.

### ***Eliminating Keyboard Contention***

When your program requires input from the keyboard, you want to make sure that Bugbyter does not interfere with your program's polling of the keyboard. Normally in Trace or Single-Step mode, Bugbyter samples (polls) the keyboard to respond when you type one of the single-keystroke commands. If you are tracing or single-stepping a program that expects input from the keyboard, your program never receives any characters unless you turn off Bugbyter's keyboard polling.

When you turn off keyboard polling, Bugbyter ignores all characters typed at the keyboard except one special *interrupt* character that you specify. Bugbyter scans the keyboard address (\$C000) during Trace or Single-Step mode, but does not clear the Keyboard-Ready flag unless the character you type is this special interrupt character. For any other characters, Bugbyter leaves these addresses intact and permits your program to accept the keystroke from the keyboard input register.

Soft switches: see the section  
"Debugging Real-Time Code."

To turn off Bugbyter's keyboard polling, you must set Bugbyter's Keyboard-Polling soft switch.

The Keyboard-Polling soft switch is located at memory location \$7C06. If you relocate Bugbyter to a different address, the Keyboard-Polling soft switch is located at the sixth location following the start of the Bugbyter. The byte stored at this location consists of

- a one-bit Keyboard-Polling flag, the most significant bit
- a seven-bit ASCII value that defines a special *interrupt* key

When you first start Bugbyter, this most significant bit is a zero. If you set this keyboard-polling flag to 1, Bugbyter turns off its keyboard polling when in Trace or Single-Step mode. With keyboard polling turned off, Bugbyter continues to scan for the special interrupt keystroke that you specified. For example, if you type

7C06:81

from the Bugbyter command level and press **RETURN**, it turns off Bugbyter's keyboard polling in Trace mode and instructs Bugbyter to scan for a **CONTROL-A** (\$01) character. The program you are debugging then can accept any keystroke from the keyboard except the **CONTROL-A** character. When you press **CONTROL-A** in Trace mode, Bugbyter exits from Trace mode and returns you to the Bugbyter command level.

#### **Using Hand Control Button 0 to Control Trace Mode**

Rather than have Bugbyter scan the keyboard during Trace mode, another way to communicate with Bugbyter in Trace mode is to use hand control button 0. This technique frees the keyboard so your own program can accept any keyboard character as input.

**0** on the Apple IIe's keyboard functions exactly as hand control button 0; if you have an Apple IIe, you can use hand control button 0 and **0** interchangeably.

To use hand control button 0 to suspend Bugbyter's Trace mode, you must activate this feature before you enter Trace mode. Pressing hand control button 0 does not cause Bugbyter to exit Trace mode, as does pressing **ESCAPE** or typing the *interrupt* key when Bugbyter's keyboard polling is turned off. Instead, when this feature is activated and you are in Trace mode, hand control button 0 suspends Bugbyter's trace of your program; Bugbyter continues tracing as soon as hand control button 0 is released.

To set up this feature, you must set Bugbyter's soft switch to \$80—normally, this soft switch is cleared or \$00. It is located at memory address 7C04 (or at start+4 if you have relocated Bugbyter to a different memory location). To set this soft switch, type

```
7C04:80
```

and press **(RETURN)**.

If you set the hand-control-button-0 soft switch to \$80 and you have not connected hand control 0 to your Apple II's GAME I/O port, your Apple II will be "frozen" if you enter Bugbyter's Trace mode. This is because your Apple II sees a disconnected hand control as a hand control with the button continually depressed.

This does not happen if you are using an Apple IIe, since **(D)** is a permanently connected *hand control button 0*.

### Using Hand Control 0

You can use hand control 0 to control Bugbyter's trace rate when you are in Trace mode. To activate this feature, simply type **(P)**. To deactivate this feature and cause Bugbyter to return to using the keyboard trace rate (shown under the *R* label in the Register subdisplay), type **(K)** while you are in Trace mode.

If you are debugging a program that itself uses hand control 0, you may want to disable this feature, causing Bugbyter to ignore a **(P)** keystroke when you are in Trace mode. To disable Bugbyter's use of hand control 0, you must clear Bugbyter's hand-control-0 soft switch, located at memory address 7C06 (or at start+6 if you relocated Bugbyter). Type

```
7C06:0
```

and press **(RETURN)**, to clear this soft switch.

To have positive control over Bugbyter's use of all three of your Apple II's input devices, you can set or clear the hand-control-button-0, hand-control-0, and the keyboard-polling soft switches all at once. To disable Bugbyter's use of the hand controls and to activate Bugbyter's keyboard polling, you can clear all three of these soft switches at once by typing

```
7C04:000
```

and pressing **(RETURN)**.




**Soft switches:** memory locations at the start of the Bugbyter program code that you can set or clear to control features of the Bugbyter.

---

### ***Executing Undefined Opcodes***

When you are debugging your program in Trace or Single-Step mode, Bugbyter ignores any illegal or undefined 6502 instruction opcodes. You can disable this restriction using a Bugbyter soft switch.

To allow Bugbyter to execute undefined 6502 opcodes, you must set the byte at relative location start + 3 (absolute location \$7C03 if the Bugbyter was loaded at address \$7C00) to \$80 (this byte is initially \$00 when Bugbyter is first loaded). To prevent Bugbyter from executing illegal 6502 instruction opcodes, you must reset this location to \$00 using Bugbyter's normal memory assignment commands.

Since Bugbyter does not know the length of an undefined opcode's operand, Bugbyter assumes no operand and increments the 6502 program counter by one. You must use  in Single-Step mode to skip past any operands to the next instruction. Using the complete register and memory display capabilities of Bugbyter, you can easily explore all of the undefined operations of the 6502 (try executing AF 58 FF, for example).



## ***The Relocating Loader***

---

<b>160</b>	Introduction to the Relocating Loader
<b>160</b>	Restrictions on Using the Relocating Loader
<b>161</b>	Using the Relocating Loader
<b>161</b>	Calling RBOOT
<b>162</b>	Using RLOAD

## The Relocating Loader

The Relocating Loader routines in this tool kit are tools that allow you to run your assembly-language subroutines from within BASIC programs, and that allow these BASIC programs to run efficiently on Apple II systems with varying amounts of memory.

There are two ways that you can run an assembly-language subroutine from a BASIC program:

- You can use the DOS BLOAD command to load your binary object file into memory, then call your routine using the fixed starting address at which your program is assembled.
- As an alternative, you can use the Relocating Loader routines to load your relocatable object program just below the Apple II's HIMEM address. This second alternative may be more attractive because it allows your BASIC program to run efficiently on Apple II systems having varying amounts of memory.

You can read more about the BLOAD command in the *DOS Programmer's Manual*.

This chapter explains how to use the Relocating Loader routines.

- The first part of this chapter explains what the Relocating Loader routines are and what they do.
- The second part of this chapter explains how you can use these routines from within a BASIC program.

You may need to read this chapter only if you want to use your assembly-language programs as part of an Applesoft BASIC program.

The Relocating Loader routines only load **relocatable** assembly-language modules (R files) that you generate using the Assembler's REL directive. REL assembly directive: see Chapter 3.

## **Introduction to the Relocating Loader**

The Relocating Loader consists of two routines, RBOOT and RLOAD. Together, these two routines allow your BASIC programs to load relocatable assembly-language subroutines into high memory, without regard to the amount of memory contained in your particular Apple II system. After your BASIC program has loaded your assembly-language subroutines, your BASIC program can call these subroutines at any time.

To use the Relocating Loader, your BASIC program must first BLOAD and then call the RBOOT routine from BASIC. This RBOOT routine loads and sets up the RLOAD routine, which performs the actual relocation of your assembly-language modules. Once you call the RBOOT routine, your BASIC program can call the RLOAD routine to load your individual relocatable subroutines into memory. After loading each relocatable subroutine into memory, the RLOAD routine returns an address at which your BASIC program can later call the actual assembly-language subroutine to perform its particular function.

When you call the RLOAD routine to load your assembly-language modules, RLOAD loads the designated module just below the HIMEM address of your Apple II system. RLOAD then reduces this HIMEM address to just below the first byte of the module that was just loaded, effectively hiding this block of code from being overwritten by Applesoft. You can load as many relocatable modules in this fashion as you like, as long as there is sufficient memory available for RLOAD and at least one free DOS file buffer.

The Relocating Loader routines do not constitute a linking loader. Although you can use these routines to load as many assembly-language modules as you like, the Relocating Loader will not resolve inter-module references or external symbols that you may have defined using the Assembler's EXTRN or ENTRY pseudo-op directives. Typically, if you use more than one assembly-language module in your BASIC program, you must call each module separately from Applesoft.

---

## **Restrictions on Using the Relocating Loader**

You can use the Relocating Loader routines with almost any BASIC program. There are several restrictions on *how* you can use these routines, and, in particular, on *when* you can use them.

Numeric variables can be used before your program calls the Relocating Loader: see the example in Figure 5-1.

- Before calling the Relocating Loader, your program must not allocate or use any string variables. The RLOAD routine does not attempt to save any string data that your program allocated before RLOAD loads relocatable modules into memory.
- After calling RBOOT, your program must not Dimension or allocate any new numeric or string variables until after the last usage of RLOAD to pull in your relocatable modules. Any variables that you use during this process must be allocated before your program calls RBOOT (see the example in Figure 5-1). This restriction is necessary since the RBOOT and RLOAD routines occupy memory just above Applesoft's variable tables. After you use RLOAD for the last time, you are free to allocate new BASIC variables that overwrite the RLOAD routine and reuse this memory space.

**Figure 5-1.** Using RBOOT and RLOAD Routines in Your BASIC Program

### **Using the Relocating Loader**

To call the Relocating Loader from a BASIC program, your program must first load and execute the RBOOT routine from Applesoft BASIC. Figure 5-1 shows the syntax of the BASIC statements that call the Relocating Loader routines.

The following example shows the syntax of the BASIC statements that call the Relocating Loader routines and load a relocatable module called MYMODULE from a disk:

```
10 ADDR = 0 : REM PRE-ALLOCATE ADDR VARIABLE

20 PRINT CHR$(4) : "BLOAD RBOOT" : CALL 520 : REM
  LOAD & CALL RBOOT

30 ADDR=USR(0) : "MYMODULE.S6.D1" : REM LOAD
  RELOCATABLE MODULE
```

Note that you cannot use the usual DS for the DOS BLOAD command; you must instead use a CHR\$(4), because you cannot allocate string variables before using the RBOOT and RLOAD routines.

### **Calling RBOOT**

The RBOOT routine is a small subroutine that you must BLOAD into addresses \$208 through \$3CF and then call by executing a CALL 520 from BASIC.

When you call RBOOT to load the RLOAD routine into memory, RBOOT loads the RLOAD routine above the end of APPLESOFT's variable tables. RBOOT accepts no parameters, but assumes that the RLOAD function is on the disk that was last accessed, which typically is the disk from which RBOOT itself was just BLOADED. The RBOOT routine then sets up the USR(0) function with the starting address of RLOAD.

### **Using the Relocating Loader**

---

### **Using RLOAD**

Once you call RBOOT, your BASIC program can call the RLOAD routine to load relocatable modules by calling the USR(0) function.

The RLOAD function takes three parameters from the statement containing the USR(0) function:

- the DOS filename of the module to be loaded
- an optional slot and drive number.

These parameters must be inside a quoted literal and may be separated from the USR function by a comma. The filename is required, although the slot and drive parameters are optional. If the filename is omitted, a `FILE NOT FOUND` error code is returned. The filename must be the name of a REL type file, not a binary file. The slot and drive parameters can appear in either order or can be omitted.

The RLOAD routine either returns the load address of the relocatable module that you loaded, or returns an error if it encounters a problem while attempting to load your module. You can catch this error by using Applesoft's ON ERR facility. If you do not use the ON ERR statement prior to using RBOOT and RLOAD, and RLOAD encounters an error while trying to load your module, your program does not function correctly.

The value returned by the USR function is a signed Real result that your program can later use to Call the loaded module. This obviously assumes that your relocatable module begins with an executable code segment.

An effective means of providing multiple entry points to your relocatable module is to put a table of jump instructions at the start of the module. This allows your BASIC program to execute CALLs to the returned `ADRS`, `ADRS+3`, `ADRS+6`, `ADRS+9`, and so forth, as a means of entering the various subfunctions in your module. This technique also allows the contents of your relocatable module to grow or shrink later and not disturb your BASIC program's interface to the assembly-language module. You can later add additional jumps to the table for new functions, while not disturbing the existing interface to your original entry points.

You can load as many modules as you wish, up to the available memory space minus the size of RLOAD itself. RLOAD is about 1.5K in length and always loaded on a page boundary by RBOOT. RLOAD requires at least one free file buffer available that it can borrow from DOS. If there is no free buffer, a NO BUFFERS AVAILABLE error occurs.

Since RLOAD reduces the address of HIMEM when it loads a module, you should save the initial value of HIMEM by Peeking it out of zero page, so that you can later restore this setting when your program finishes executing. The Relocating Loader does not automatically restore HIMEM to its original setting.



---

**Warning**

When you are testing a program that uses the Relocating Loader, you should be aware that repeated use of RLOAD without restoring HIMEM causes RLOAD to allocate new space each time that it loads a module. This can eat up all of memory in short order if you do not restore HIMEM to its normal value before each test. An easy way to test such a program is to issue an FP command to DOS, and then Run the program from the disk. Be aware that the FP command erases the program currently in memory. This can cause you to execute extra Saves during the testing process, but saving your program occasionally is a good practice anyway.

---





## ***The Apple IIe System Identification Routines***

- 
- 168** Introduction to the Identification Routine
  - 168** Using the Routines in Your Assembly-Language Programs
  - 169** Overview of the Identification Procedure

## The Apple IIe System Identification Routines

If you are only writing programs that you will use yourself, you may not need to read this chapter.

The Apple IIe System Identification routines in this tool kit allow your assembly-language and BASIC programs to identify the type of Apple II system on which they are running. If you intend to write programs that might eventually run on different Apple II systems, then you might consider using these routines in your program.

When you incorporate a System Identification routine into your program, your program can identify whether it is currently running on an Apple II, an Apple II Plus, or an Apple IIe computer. If your program is running on an Apple IIe system, these routines also identify the additional accessory cards that are currently installed, such as the Apple IIe 80-Column Text Card, or the Apple IIe Extended 80-Column Text Card.

Using this information, your programs can take advantage of all the capabilities of the more powerful Apple II systems, while still maintaining compatibility with the whole family of Apple II computer systems.

This chapter explains what the Identification Routines can do, and describes how you can use these routines in your assembly-language programs. It specifically

- introduces the Identification Routines and explains the information that this routine can return to your program;
- explains how you can incorporate these routines into your assembly-language programs;
- gives an overview of the identification procedure, if you want only to incorporate portions of the identification routines in your program. For your convenience in customizing or expanding the identification routines, assembly-language source code for the identification routines is included in this tool kit. This section serves as documentation for the accompanying source code.

For information about using the Identification Routines in BASIC programs, see the corresponding chapter in the *Applesoft/DOS Programmer's Tool Kit*.

## **Introduction to the Identification Routine**

The Apple IIe System Identification Routine provides a standard way of identifying the various configurations of the Apple II. These routines are intended for both professional and amateur programmers: anyone who is writing programs that might run on more than one Apple II system. Since your particular program may not need all the parts of the complete identification routine, this tool kit includes the source code for both the assembly-language and BASIC versions of this program.

You have a number of files on the DOS Programmer's Tool Kit Volume II disk related to the Apple II System Identification routines. These files are

- All ASSEMBLY ID.S—the assembly-language source code for the Apple II System Identification routine
- All ASSEMBLY ID—the machine-language (binary) version of these routines.

When you call the identification routine from your assembly-language program, the identification routine returns a single value that identifies the type of Apple II system on which the program is executing and the capabilities of that system if it is an Apple IIe. Table 6-1 shows the possible values that may be returned and the corresponding machine configurations that these values indicate.

**Table 6-1.** Identification Procedure Return Values

Returned Value	Your System Configuration
\$00	An Apple II or Apple II Plus
\$20	An Apple IIe that does not have an 80-Column Text Card
\$40	An Apple IIe with an 80-Column Text Card
\$80	An Apple IIe with an Extended 80-Column Text Card

## **Using the Routines in Your Assembly-Language Programs**

The most efficient way to incorporate the Apple II System ID routines into your assembly-language programs is simply to include the All ASSEMBLY ID.S source file into your own program source file using the Assembler's INCLUDE directive. A second alternative is to use the Editor to append your program's source file with the All ASSEMBLY ID.S source statements to create a single file containing both modules. In either case, when

you assemble your source file, you automatically incorporate the Apple II System ID routines into your program. Only slight modifications to the AII ASSEMBLY ID.S source text are required to make this module a subroutine within your main program.

The binary code file AII ASSEMBLY ID is a machine-language subroutine that starts at location \$2D4 and occupies memory from location \$2D0 to \$3CF. If you are not using the Editor/Assembler to develop your application program, there is yet another way to link the Apple II System ID routine into your program. If you BLOAD both your main program and then the AII ASSEMBLY ID (machine-language version), you can then BSAVE both modules together on the same disk file. Your main program must call the identification routine using a JSR to location \$2D4, and then retrieve the identification code from location \$3CF once the identification routine returns control to your calling program.

### **Overview of the Identification Procedure**

This section contains an overview of the procedure that the identification routines go through to identify the particular Apple II configuration on which your program is running.

For some programs, you may not need to identify the exact configuration of your computer. For example, your program may not use auxiliary memory and you may not care to waste code space or execution time to find out if this memory is installed. For these reasons, you may wish to modify these identification routines. If you do this, however, Apple PCS Product Marketing Technical Support requests that your altered programs determine the Apple II configurations in the same way as the identification routines included here. Future revisions of the Apple IIe may not be compatible with other identification procedures.

The following is a list of the steps carried out by the Apple IIe identification routine as it determines the configuration of your Apple II system:

1. Save four identification bytes from the ROM/RAM area (\$D000 to \$FFFF).
2. Disable interrupts.
3. Switch bankable memory to read ROM by reading \$C089 twice.
4. Identify Apple IIe by finding the number 6 at \$FBB3.
5. If it is an Apple IIe and the high bit is on at location \$C017, then it has an 80-column text card.

6. If it is an Apple IIe with an 80-column text card, then check for auxiliary memory:
  - If \$C013's high bit is on, then it is reading auxiliary memory so it must have auxiliary memory.
  - If \$C016's high bit is on, then it is reading alternate zero page so it must have auxiliary memory.
  - If sparse memory mapping (no upper four address bits so that \$800 has the same RAM location as \$C00), then it has no auxiliary memory.
    - a. Exchange a section of zero page with the section of code that switches memory banks. This way the zero page data is saved and the program doesn't get switched out.
    - b. Jump to the relocated code on zero page.
    - c. Switch in auxiliary memory (\$200-\$BFFF) for reading and writing by writing to \$C005 and \$C003. Note: Auxiliary memory locations \$400-\$800 and \$2000-\$4000 may not be available depending upon the setting of soft switches for 80-column video display and high-resolution graphics; they have priority over auxiliary memory selection.
    - d. Store a value at \$800 and see if the same value is at \$C00; if not, then there is auxiliary memory.
    - e. Change the value at \$C00 and see if \$800 changes to the same value; if so, then there is no auxiliary memory.
    - f. Set soft switches for reading and writing to main memory by writing to \$C002 and \$C004.
    - g. Jump back into program on main RAM.
    - h. Put zero page back.
7. Store identification byte for later reference by calling routine.
8. Restore the RAM/ROM area to its original state by checking four bytes saved at the start of the routine.
9. Enable interrupts.
10. Return to calling program.



*[Faint, illegible text visible through the paper, likely bleed-through from the reverse side.]*

## ***Appendixes***

### ***A***

#### ***An Editor Quick Guide***

**175**

- 175 Editor Commands: A Functional Summary
- 175 Executing Direct DOS Commands
- 175 Storing and Retrieving Files From a Disk
- 176 Manipulating Lines in the Text Buffer
- 176 Viewing Your Text in the Text Buffer
- 177 Changing Text Within a Line
- 177 Editing Two Files at Once
- 177 Altering the Display
- 178 Leaving the Editor
- 178 Calling the Assembler
- 179 Editor Commands: An Alphabetic Summary
- 182 Edit Mode Keystroke Summary

### ***B***

#### ***6502 Assembly Language Summary***

**183**

- 183 Mnemonic Summary
- 185 Addressing Mode Summary
- 185 Assembler Directive Summary

### ***C***

#### ***A Bugbyter Quick Guide***

**187**

- 187 Bugbyter Command Level
- 187 General Commands
- 187 Command-Line Editing
- 188 Memory Reference
- 189 Register Reference
- 189 Trace/Single-Step Mode
- 190 Disassembly Options for Trace/Single-Step
- 190 Breakpoints
- 191 Debugging in Execution Mode
- 191 User Soft Switches



**D**

**Error Messages**

**193**

- 193 Editor Error Messages
- 193 Editor's DOS-Error Messages
- 195 Editor Command-Error Messages
- 197 Assembler Error Messages
- 197 Assembler's DOS-Error Messages
- 199 Assembler Syntax-Error Messages

**E**

**Object File and Symbol Table Formats**

**207**

- 207 Object File Format
- 209 Symbol Table Formats

**F**

**Editing BASIC Programs**

**211**

- 211 Using the Editor to Edit BASIC Programs

**G**

**The SWEET16 Interpreter**

**213**

- 213 A 16-Bit Pseudomachine
- 214 SWEET16 Interpreter Operation
- 215 SWEET16 Instruction Summary
- 215 Register Instructions
- 215 Control Instructions

**H**

**System Memory Usage**

**217**

- 217 The Editor/Assembler
- 218 The Bugbyter Debugger

**I**

**Editor/Assembler File Components**

**219**

- 219 The EDASM Files
- 220 Making a HELLO Program

## An Editor Quick Guide

### Editor Commands: A Functional Summary

Editor Command Form	Command Description
<b>Executing Direct DOS Commands</b>	
.doscommand	Issues the characters after the period to DOS 3.3 to be processed as a DOS command. DOS checks the syntax and issues any required error messages.
<b>Storing and Retrieving Files From a Disk</b>	
LOAD fname [,Ss] [,Dd] [,Vv]	Loads the file with fname into the text buffer, destroying anything already in the buffer.
APPEND [line#] fname [,Ss] [,Dd] [,Vv]	Appends all the lines of fname to the text buffer after [line#].
SAVE [begin# [-end#]] [fname] [,Ss] [,Dd] [,Vv]	Saves the lines [in the indicated range] to the file named fname.
CATalog [,Ss] [,Dd] [,Vv]	Displays the DOS catalog of the disk in the current SSlot, DDrive, and Volume.
FILE	Displays the current status of the Editor's filename, SSlot, DDrive, and Volume parameters along with the text buffer memory usage totals.
LENgth	Displays the current text buffer's file usage totals.

Slot slot#	Sets the current disk SLOt to slot#.
DRive drive#	Sets the current disk drive to drive#.
Volume vol#	Sets the current volume number to vol#.

---

### ***Manipulating Lines in the Text Buffer***

Add [line#]	Adds lines [after line#] to the text buffer. Enters Input mode to add the new lines.
Insert line#	Inserts lines from the keyboard before line#. Enters Input mode to insert the new lines.
COpy line#1 [-line#2] TO line#3	Copies line#1 [thru line#2] to before line#3.
Delete begin# [-end#]	Deletes the range of lines from the buffer.
Replace begin# [-end#]	Deletes the range of lines and enters Input mode, like Insert, to add new lines to the text buffer in place of the deleted lines.
NEW	Deletes the entire contents of the text buffer.

---

### ***Viewing Your Text in the Text Buffer***

List [begin# [-end#]]	Displays, with line numbers, the lines in the indicated range within the text buffer. Control characters display in inverse video.
<b>CONTROL-R</b>	Relists the lines of text listed by the most recent List command.
Print [begin# [-end#]]	Outputs, without line numbers, the lines within the indicated range. Control characters are output as control characters.

---

### Changing Text Within a Line

Find [begin# [-end#]] [.string.]	Displays all lines within the indicated range that contain the string.
Change [begin# [-end#]] .oldstr.newstring.	Changes All/Some of the occurrences of a string (oldstr) to a new string (newstr) within the range of lines indicated.
Edit [begin# [-end#]] [.string.]	Enters Edit mode for all lines indicated [that contain the string].
SET Delim .char.	Changes the Editor command delimiter character to the new character indicated. This permits the colon character to be used in a search string.

---

### Editing Two Files at Once

SWAP	Swaps the currently-active text buffer, saving the contents of the current one for later editing.
KILL2	Deletes the entire contents of text buffer 2 and returns to single text buffer mode with text buffer 1.

---

### Altering the Display

COLumn 40	Sets the Editor display routines to 40-column display mode.
COLumn 80	Sets the Editor display routines to 80-column display mode, if your Apple II can support this.
<b>CONTROL</b> -E	Enables lowercase character entry (only for Apple II or Apple II Plus systems that cannot otherwise receive lowercase characters).
<b>CONTROL</b> -W	Disables lowercase character entry that was enabled by <b>CONTROL</b> -E command.

SET Lcase	Enables lowercase character entry (for Apple II or Apple II Plus systems that do not contain an ALS Smarterm Card, but can receive and display lowercase characters).
SET Ucase	Disables lowercase character entry that was enabled by the SET Lcase command.
Tabs [Tablist] [.tabchar.]	Sets the Editor display tabs stops to the Tablist and sets the tab character to tabchar.
TRuncON	Turns on comment display truncation.
TRuncOFF	Turns off comment display truncation.

---

### ***Leaving the Editor***

END	Ends the Editor session and returns control to the BASIC interpreter.
MON	Enters the Apple II Monitor. (Type <b>CONTROL</b> - <b>Y</b> to return to the Editor.)
Where line#	Displays the hexadecimal address of text for line# (address useful when using Monitor commands).

---

### ***Calling the Assembler***

PR# slot# [,DevCtrlstrg]	Sets the Assembler output device to slot (slot#) and saves the DevCtrlstrg for device initialization by the Assembler.
ASM srcfile [, @, objfile [,Ss [,Dd [,Vv]]]	Calls the Assembler to assemble the source file (srcfile) and create an object file (objfile) on Slot s, Drive d, Volume v.

## Editor Commands: An Alphabetic Summary

Editor Command Form	Command Description
Add [line#]	Adds lines [after line#] to the text buffer. Enters Input mode to add the new lines.
APPEND [line#] fname [,Ss] [,Dd] [,Vv]	Appends all the lines of fname to the text buffer after [line#]
ASM srcfile [, @, objfile [,Ss [,Dd] [,Vv]]]	Calls the Assembler to assemble the source file (srcfile) and create an object file (objfile) on Slot s, Drive d, Volume v.
CATalog [,Ss] [,Dd] [,Vv]	Displays the DOS catalog of the disk in the current SLOt, DRive, and Volume.
COLumn 40	Sets the Editor display routines to 40-column display mode.
COLumn 80	Sets the Editor display routines to 80-column display mode, if your Apple II can support this.
(CONTROL)-(E)	Enables lowercase character entry (only for Apple II or Apple II Plus systems that cannot otherwise receive lowercase characters).
(CONTROL)-(R)	Relists the lines of text listed by the most recent List command.
(CONTROL)-(W)	Disables lowercase character entry enabled by (CONTROL)-(E) command.
Change [begin# [-end#]] .oldstr.newstring.	Changes All/Some of the occurrences of a string (oldstr) to a new string (newstr) within the range of lines indicated.
COPy line#1 [-line#2] TO line#3	Copies line#1 [thru line#2] to before line#3.
Delete begin# [-end#]	Deletes the range of lines from the buffer.

.DOS Command	Issues the characters after the period to DOS 3.3 to be processed as a DOS command. DOS checks the syntax and issues any required error messages.
DRive drive#	Selects the current disk drive for use by the Editor disk I/O commands.
Edit [begin# [-end#]] [.string.]	Enters Edit mode for all lines indicated [that contain the string].
END	Ends the Editor session and returns control to the BASIC interpreter.
FILE	Displays the current status of the Editor's filename, SLoT, DRive, and Volume parameters along with the text buffer memory usage totals.
Find [begin# [-end#]] [.string.]	Displays all lines within the indicated range that contain the string.
Insert line#	Inserts lines from the keyboard before line#. Enters Input mode to insert the new lines.
KILL2	Deletes the entire contents of text buffer 2 and returns to single text buffer mode with text buffer 1.
LENgth	Displays the current text buffer's file usage totals.
List [begin# [-end#]]	Displays, with line numbers, the lines in the indicated range within the text buffer. Control characters display in inverse video.
LOAD fname [,Ss] [,Dd] [,Vv]	Loads the file with fname into the text buffer, destroying anything already in the buffer.
MON	Enters the Apple II Monitor. (Type <b>CONTROL</b> - <b>Y</b> to return to the Editor.)
NEW	Deletes the entire contents of the text buffer.



Print [begin# [-end#]]	Outputs, without line numbers, the lines within the indicated range. Control characters are output as control characters.
PR# slot# [,DevCtrlstrg]	Sets the Assembler output device to slot (slot#) and saves the DevCtrlstrg for device initialization by the Assembler.
Replace begin# [-end#]	Deletes the range of lines and enters Input mode, like Insert, to add new lines to the text buffer in place of the deleted lines.
SaVE [begin# [-end#]] [fname] [,Ss] [,Dd] [,Vv]	Saves the lines (in the indicated range) to the file named fname.
SET Delim .char.	Changes the Editor command delimiter character to the new character indicated. (Permits the colon character to be used in a search string).
SET Lcase	Enables lowercase character entry (for Apple II or Apple II Plus systems that do not contain an ALS Smarterm Card, but can receive and display lowercase characters).
SET Ucase	Disables lowercase character entry that was enabled by the SET Lcase command.
SLot slot#	Sets the current disk SLot to slot#.
SWAP	Swaps the currently-active text buffer, saving the contents of the current one for later editing.
Tabs [Tablist] [.tabchar.]	Sets the Editor display tabs stops to the Tablist and sets the tab character to tabchar.
TRunCOff	Turns off comment display truncation.
TRunCOOn	Turns on comment display truncation.

Volume vol#

Sets the current volume number for use in the disk I/O commands.

Where line#

Displays the hexadecimal address of text for line#.

### ***Edit Mode Keystroke Summary***

#### **Editing Function**

#### **Edit Mode Keystroke**

Move cursor **left** one character

**←**

Move cursor **right** one character

**→**

**Delete** current character

**CONTROL-D**

**Insert** characters at this position

**CONTROL-I**

**Replace** a character

any noncontrol character

Enter control character into text (**Verbatim**)

**CONTROL-V**, any character

**Restore** the original line

**CONTROL-R**

**Find** the indicated character in the line and move the cursor there

**CONTROL-F**, any character

**Store** line as it appears on screen

**RETURN**

**Truncate** line after current cursor and store the remainder in text buffer

**CONTROL-T**

**Cancel** Edit mode and return to command level

**CONTROL-X**

## 6502 Assembly Language Summary

### Mnemonic Summary

Mnemonic	Instruction
ADC	$A + M + C \rightarrow A$
AND	$A \text{ and } M \rightarrow A$
ASL	$C \leftarrow [7..0] \leftarrow 0$
BCC	Branch on $C = 0$
BCS	Branch on $C = 1$
BEQ	Branch on $Z = 1$
BGE	Branch on $C = 1$
BIT	$A \text{ and } M, M7 \rightarrow N, M6 \rightarrow V$
BLT	Branch on $C = 0$
BMI	Branch on $N = 1$
BNE	Branch on $Z = 0$
BPL	Branch on $N = 0$
BRK	Force Break
BVC	Branch on $V = 0$
BVS	Branch on $V = 1$
CLC	$0 \rightarrow C$
CLD	$0 \rightarrow D$
CLI	$0 \rightarrow I$
CLV	$0 \rightarrow V$
CMP	$A - M \text{ status} \rightarrow P$
CPX	$X - M \text{ status} \rightarrow P$
CPY	$Y - M \text{ status} \rightarrow P$
DEC	$M - 1 \rightarrow M$
DEX	$X - 1 \rightarrow X$
DEY	$Y - 1 \rightarrow Y$
EOR	$A \text{ xor } M \rightarrow A$
INC	$M + 1 \rightarrow M$
INX	$X + 1 \rightarrow X$
INY	$Y + 1 \rightarrow Y$

JMP	Jump to New Location
JSR	Jump to Subroutine
LDA	M -> A
LDX	M -> X
LDY	M -> Y
LSL	C <- [7..0] <- 0
LSR	0 -> [7..0] -> C
NOP	No Operation (PC=PC+1)
ORA	M or A -> A
PHA	A -> Ms S-1 -> S
PHP	P -> Ms S-1 -> S
PLA	S+1 -> S Ms -> A
PLP	S+1 -> S Ms -> P
ROL	$\overline{C} \leftarrow [7..0] \leftarrow C \leftarrow \overline{C}$
ROR	$\overline{C} \rightarrow C \rightarrow [7..0] \rightarrow \overline{C}$
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	A - M - C -> A
SEC	1 -> C
SED	1 -> D
SEI	1 -> I
STA	A -> M
STX	X -> M
STY	Y -> M
TAX	A -> X
TAY	A -> Y
TSX	S -> X
TXA	X -> A
TXS	X -> S
TYA	Y -> A

where

A, X, Y, S, and P are the 6502 registers

M is a memory location

C is the Carry bit of the P-register

+ indicates binary addition

[7..0] is a bit description of M or A

Ms indicates the location to which the S-register points.

See the Preface, section "Reference  
Manuals on the 6502  
Microprocessor."

## Addressing Mode Summary

Note that all required syntax may be preceded by an optional identifier in the label field of a statement.

Addressing Mode	Required Syntax
Implied (no address)	opc
Accumulator	opc A
Immediate	opc #expression
Low 8 bits of address	opc #>expression
High 8 bits of address	opc #<expression
Zero page	opc zpg-expression
Indexed X	opc zpg-expression,X
Indexed Y	opc zpg-expression,Y
Absolute	opc abs-expression
Indexed X	opc abs-expression,X
Indexed Y	opc abs-expression,Y
Indexed,Indirect X	opc (zpg-expression),X
Indirect,Indexed Y	opc (zpg-expression),Y
Absolute Indirect	JMP (abs-expression)

where

opc refers to an instruction mnemonic  
abs refers to an absolute address expression  
zpg refers to a zero-page address expression.

All other characters must be typed as shown.

**By the Way:** In zero-page address expressions, the expression result must either be less than 256, or you must use the low-byte operator (>) in front of the expression to indicate a single-byte result.

This is only a summary. Please refer to the *6502 Programming Manual*, published by the manufacturers of the 6502 microprocessor, for complete details.

## Assembler Directive Summary

Directive Syntax Format	Directive Description
ASC .string.	ASCII character data
CHN filename [,slot][,drive][,vol]]	Chain to new source file
CHR /?	Character used for REPEAT
DATE	DATE character data (9)
DCI .string.	special ASCII character data

DDB expr[,expr...]	Define Double Byte
DEF identifier	Define absolute identifier
DEND	Dsect END
DFB expr[,expr...]	DeFine Byte(s)
DO expr	DO assembly if expr > 0
DS expr[,expr]	Define Storage [value]
DSECT	Dummy Section beginning
DW expr[,expr...]	Define Word(s)
ELSE	complement Assembly mode
ENTRY identifier	define ENTRY identifier
identifier EQU expr	EQUate identifier to expr
EXTRN identifier	Refer to external identifier
FAIL p,.string.	FAILure error message
FIN	FINish conditional assembly
IBUFSIZ expr	Set INCLUDE-buffer size
IDNUM	generate IDNUM data (6 bytes)
IFEQ expr IFGE expr	begin conditional assembly
IFGT expr	
IFNE expr IFLE expr	begin conditional assembly
IFLT expr	
INCLUDE	INCLUDE file into source
filename[,slot][,drive][,vol]]	
LST [ON,OFF]	Control listing options
[,NO]opt[,NO]opt...]]	
MACLIB [slot][,drive][,vol]]	Macro library disk enable
MSB [ON , OFF]	Most Significant Bit set
OBJ expr	OBJect memory address set
ORG expr	Origin of Assembler adrs
PAGE	eject a PAGE on listing
REF identifier	REFERence global identifier
REL	generate RELOcatable output
REP expr	Repeat CHR expr times
SBTL .string.	define subtitle string
SBUFSIZ expr	Set source-buffer size
SKP expr	SKip expr blank lines
STR .string.	counted ASCII string data
SW16 [expr]	call SWEET16 interpreter
ZDEF identifier	Zero-page global definition
ZREF identifier or ZXTRN	Zero-page external reference
identifier	

## A Bugbyter Quick Guide

### Bugbyter Command Level

#### General Commands

.doscommand	Execute DOS command. Press <b>RETURN</b> to return to Bugbyter.
<b>Q</b>	Quit Bugbyter (exits through DOS vector \$3D0).
<b>M</b>	Enter Monitor. Return to Bugbyter with <b>CONTROL-Y</b> .
addressL	Disassemble object code beginning at address.
<b>L</b>	Continue disassembling object code.
SET	Customize Bugbyter's Master Display, where:
	<b>↓</b> Moves window down.
	<b>↑</b> Moves window up.
	<b>RETURN</b> Fixes subdisplay and advances to next subdisplay.
ON	Turn Bugbyter's Master Display on.
OFF	Turn Bugbyter's Master Display off.
<b>V</b>	Display copyright and version number.

#### Command-Line Editing

Keystroke	Editing Function
<b>RETURN</b>	Accept user-entered command line.
<b>←</b>	Move cursor to the left.
<b>→</b>	Move cursor to the right.
<b>CONTROL-B</b>	Move cursor to beginning of command line.
<b>CONTROL-C</b>	Accept next keystroke verbatim.
<b>CONTROL-D</b>	Delete a character.



**CONTROL-I**

Enter insert-character mode (any other function key exits from this mode).

**CONTROL-N**

Move cursor to end of command line.

**CONTROL-X**

Delete command line.

---

### Memory Reference

MEM

Edit memory subdisplay where:

**H** Display contents of address as hex and ASCII, or

**P** Display contents of address and address + 1 as pointer.

address Enter hex address of memory to display.

**→** Advance to next cell.

or **SPACE**

or **RETURN**

**←** Return to previous cell.

**ESCAPE** Return to Bugbyter command line.

address:mnemonic

Bugbyter assembles mnemonic, placing opcode at address.

or address:value

Fill address with hex value.

or address:"text"

Fill address with ASCII character (MSB on).

or address:'text'

Fill address with ASCII character (MSB off).

or (any mixture)

Multiple values and ASCII text (MSB on or off) can be mixed freely in memory fill. Type slash (/) to enter the next character verbatim.

**SPACE**

Display next available address, allowing you to type any of the above commands.

address:

Enter Memory Page display; showing 184 memory cells (starting at address) in hex and ASCII. Press **SPACE** and then press **RETURN** to display next 184 cells. Press **ESCAPE** to return to Bugbyter command level and Master Display.

+decimalvalue=

Convert positive decimal to hex.

-decimalvalue=

Convert negative decimal to hex (65536-decimalvalue).

value=

Convert hex to decimal.

\$value=

Convert hex to decimal.

---

### Register Reference

PC=address	Set 6502 Program Counter with hex address.
A=value	Set 6502 A-register with hex value.
X=value	Set 6502 X-register with hex value.
Y=value	Set 6502 Y-register with hex value.
S=value	Set 6502 Stack Pointer with hex value.
P=value	Set 6502 Processor Status register with hex value.
C=value	Set Bugbyter Cycle Counter to value.
R=value	Set Bugbyter keyboard Trace Rate to value.

---

### Trace/Single-Step Mode

From Bugbyter's command level, you can enter Single-Step mode by typing:

addressS	Enter Single-Step mode and execute single opcode starting at address.
(S)	Enter Single-Step mode, continuing from previous address.

From Bugbyter's command level, you can enter Trace mode by typing:

addressT	Enter Trace mode and interpret instructions starting at address.
(T)	Enter Trace mode, continuing from previous address.

Once in Single-Step or Trace mode (see S or T commands above), you can enter any of the following single-keystroke commands:

(SPACE)	Single-step, executing next instruction only.
(→)	Skip next instruction.
(RETURN)	Enter Trace mode for continuous trace.
(R)	Trace until RTS opcode encountered.
(ESCAPE)	Return to Bugbyter command level.
(C)	Clear Cycle Counter.
(P)	Use hand control 0 to adjust Trace Rate.
(K)	Use Keyboard Rate (R=value) to adjust Trace Rate.
(Q)	Sound off (Quiet).
(S)	Sound on.

<input type="checkbox"/> 1	Display primary Apple II screen.
<input type="checkbox"/> 2	Display secondary Apple II screen.
<input type="checkbox"/> T	Display Apple II Text screen.
<input type="checkbox"/> L	Display Apple II Low-resolution graphics screen.
<input type="checkbox"/> H	Display Apple II High-resolution graphics screen.
<input type="checkbox"/> F	Display Full screen graphics.
<input type="checkbox"/> M	Display Mixed text and graphics.

---

### **Disassembly Options for Trace/Single-Step**

From the Bugbyter command level, you can select one of the following options:

O=A	Display 6502 Accumulator in binary.
O=X	Display 6502 X-register in binary.
O=Y	Display 6502 Y-register in binary.
O=S	Display 6502 Stack Pointer in binary.
O=P	Display 6502 Processor Status register in binary.
O=B	Display instruction bytes in hex.
O=E	Display computed effective addresses or relative branches and instruction cycles.

---

### **Breakpoints**

All breakpoints are set and cleared from the Bugbyter command level.

BPn	Set breakpoint "n" where:
value	Sets breakpoint field to value.
<input type="button" value="←"/>	Moves to previous field.
<input type="button" value="→"/>	Moves to next field.
or <input type="button" value="SPACE"/>	
<input type="button" value="ESCAPE"/>	Returns to Bugbyter command line.
or <input type="button" value="RETURN"/>	
	POINT is definable breakpoint address.
	COUNT is the number of times Bugbyter encountered the breakpoint address.

TRIG is a definable count before breaking. Note: To cause a break, TRIG must be one or greater.  
BROKE is the number of times Bugbyter triggered.

IN	Insert real breakpoints (BRK opcodes (00)) into addresses specified in breakpoint subdisplay. Disables breakpoint modification.
OUT	Replace BRK opcodes with original instructions at addresses specified in breakpoint subdisplay. Enables breakpoint modification. Used for interpretive debugging (default).
CLR	Clear all breakpoints.
CLRn	Clear breakpoint "n".

### **Debugging in Execution Mode**

From the Bugbyter command level, you can enter Execution mode by typing one of the following commands:

addressG	Enter Execution mode, executing code as a subroutine starting at address.
(G)	Execute code as subroutine, continuing from previous address. An RTS opcode returns to Bugbyter.
addressJ	Enter Execution mode, jumping to code at address.
(J)	Enter Execution mode, continuing from previous address.

### **User Soft Switches**

Location	Soft Switch Function
start + 3	Execute undefined 6502 opcodes (\$80 = on; 0 = off — default).
start + 4	Use hand control button 0 to suspend Trace (\$80 = on; 0 = off — default).
start + 5	Use hand control 0 for Trace Rate (\$80 = on — default; 0 = off).

start + 6	Disable keyboard polling in Trace, Single-Step modes (MSB on + ASCII character code for escape character, MSB off = normal polling—default).
start + 7	Sound (\$80 = on—default; 0 = off).
start + 8, + 9	Cycle Counter value (low-byte, high-byte).
start + \$A, + \$B	Starting address of real-time code (default = \$FFFF).
start + \$C, + \$D	Ending address of real-time code (default = \$FFFF).
CALL 1016	Restart Bugbyter using the Monitor's <u>CONTROL</u> -(Y) vector.
CALL 8192	Restart Bugbyter using Bugbyter's specific load address.

## **Error Messages**

---

### **Editor Error Messages**

Two types of errors may occur when you are using the Editor:

- DOS errors, indicating that a problem occurred while reading your source file from disk, or while saving your edited text to a disk.
- Editor command errors, indicating that you typed an unrecognized command or that you typed some command parameter incorrectly.

If a DOS error occurs while you are using the Editor, the Editor displays a normal DOS error message and allows you to correct the problem. If the Editor finds an error in a command you typed, the Editor displays an appropriate error message and ignores the illegal command. In neither case do you leave the Editor or lose your edited text simply because an error occurred. The error messages that you may see in both cases are discussed in the sections that follow.

---

### **Editor's DOS-Error Messages**

The Editor/Assembler uses DOS to manipulate files that are stored on a disk. In doing so, several DOS errors may commonly occur. When a DOS error occurs while you are using the Editor, the Editor displays a DOS-error message and returns to the Editor command level, allowing you to correct the problem and retype the command that resulted in the DOS error.

The descriptions that follow show some of the ways in which DOS errors can happen. It presents the most common errors, but does not present the total list of errors that can occur.

#### **WRITE PROTECTED**

A **WRITE PROTECTED** message indicates that you attempted to save to a disk that has a write-protect tab, or that you typed the **ASM** command when your source disk was write-protected. This message appears following the **ASM** command because a write-protected source disk prevents the Editor/Assembler from updating the **ASMIDSTAMP** file that tracks your source disks.

#### **FILE NOT FOUND**

The **FILE NOT FOUND** message occurs immediately after the **ASM** command if you do not have a file named **ASMIDSTAMP** on your program source disk. The Editor/Assembler does not require that this file be present, so you should ignore this message. This error can also occur if you try to issue a Direct DOS command, such as **Lock** or **Unlock**, that cannot find a file.

#### **I/O ERROR**

An **I/O ERROR** message typically indicates a bad disk media or a disk that is improperly inserted into the drive. If this error occurs while you are saving a file, your output disk is probably bad and you should use the **Save** command with another disk. If it occurs during execution of a **Load** command, your file was only partially read in and may be permanently lost (backing up is wise). If this error occurs while you are using the **Catalog** command, you're in deep trouble and may have lost your disk directory.

#### **DISK FULL**

A **DISK FULL** error usually occurs during a **Save** command: if you get this message, you should save your file onto another disk.

#### **FILE LOCKED**

The **FILE LOCKED** error can occur if you are in the habit of locking your files and try to save to a file that is locked. The locked file remains intact, as does your current edit file. Unlock the file or change the filename before trying to save the file again. It is excellent practice to lock all the files on your disk except the one you are editing to avoid losing a file because you saved your edited file with the wrong name.

#### **SYNTAX ERROR**

The **SYNTAX ERROR** message is the result of issuing a direct DOS command with incorrect syntax.



#### **NO BUFFERS AVAILABLE**

This can occur if you call the system after MAXFILES has been set to one and you somehow issue a DOS command that tries to use two files at once. The system initially allocates five files, but pressing **RESET** can cause files to be left open. If you issue a direct DOS **.CLOSE** command, the unused open files can be recovered.

#### **FILE TYPE MISMATCH**

This occurs if you try to load a file other than a text file or try to save using a name that is already in use by a another type of file, such as integer or binary.

#### **PROGRAM TOO LARGE**

If this occurs, you tried to do a direct DOS Load command and have clobbered the Editor/Assembler, along with your edit file. Read the warnings regarding direct DOS commands in the chapter on the Editor.

---

### **Editor Command-Error Messages**

The Editor checks the syntax and parameters of each command that you type to ensure that it is valid. The command descriptions in Chapter 2 include the specific error messages that may result if you type a command incorrectly.

#### **ERR: BAD FORMAT**

The **ERR: BAD FORMAT** message occurs if you type the first string field of the Change command as a null string.

#### **ERR: BAD RANGE**

The **ERR: BAD RANGE** message occurs if the second line number in a range (begin#-end# or begin#-count) has an invalid ending line number. The Copy command requires the second line number of its range parameter to be an existing line in the Editor's text buffer.

#### **ERR: BAD SLOT/DRIVE/VOL #**

This error message indicates that a Slot, Drive, or Volume command has an invalid parameter. Valid slot numbers are 0 through 7. Valid drive numbers are 1 and 2. Valid volume numbers are 0 through 254.

#### ***BUFFER ERROR***

A **BUFFER ERROR** message indicates that you typed the **ASM** command while you still had text in the Editor's text buffer. Before using the Assembler, save this text and then clear the text buffer using the **New** command. The Editor signals this error to prevent you from accidentally overwriting your text by calling the Assembler without first saving your text.

#### ***CMD SYNTAX ERROR***

The **CMD SYNTAX ERROR** indicates that you typed extra characters following a valid command or command parameter.

#### ***ERR: MEMORY FULL***

The **ERR: MEMORY FULL** message occurs when you fill all available text buffer space. This error appears when the Editor attempts to insert a line of text into the full buffer; when this happens you lose the line of text that you just typed or edited. The **Load** and **Append** commands also result in this error if the file being read overflows the text buffer.

#### ***MULTI BUFFER ERROR***

The **MULTI BUFFER ERROR** signals that you attempted to use the **ASM** command when both Editor buffers are active. See the **Kill2** command.

#### ***NUMERIC OVERFLOW ERROR***

A **NUMERIC OVERFLOW ERROR** results if a line number has a value larger than 63999.

#### ***PARAMETER(S) OMITTED ERROR***

The **PARAMETER(S) OMITTED ERROR** message indicates that you neglected to type a necessary parameter for the command you just entered.

#### ***ERR: SYNTAX***

A **ERR: SYNTAX** message following a **Delete** or **Replace** command indicates that you typed a range of line numbers with an implied ending line number. The same message following the **Save** command indicates that you must type a filename parameter; no filename was previously defined by a **Load** or **Save** command.

### **UNKNOWN COMMAND ERROR**

The UNKNOWN COMMAND ERROR indicates that you typed a command name that is misspelled or not recognized by the Editor.

---

## **Assembler Error Messages**

Two types of errors may occur when you are using the Assembler:

- DOS errors indicate that a problem occurred while reading your source program from a disk, or while writing your object code to a disk. When a DOS error occurs, the Assembler immediately stops the assembly process.
- Assembler syntax errors indicate that the Assembler encountered an unrecognized syntax or illegal usage in one of the assembly statements in your source file. When the Assembler encounters a syntax error, it prints an appropriate error message and continues with the assembly process.

Both types of error messages are discussed in the sections that follow.

---

### **Assembler's DOS-Error Messages**

When a DOS error occurs during an assembly, the Assembler stops the assembly, displays the DOS error message to indicate the problem, and prints the following message on the screen:

ASSEMBLY ABORTED. PRESS RETURN

The Assembler then waits for you to press **RETURN** before it attempts to close any open files. If the Assembler encounters another DOS error while trying to close files, the Assembler returns you to the Editor command level.

The DOS error messages are also explained in the *DOS User's Manual* in the section on DOS messages. Many of these explanations are equally applicable to situations in which you use the Assembler; you may find it helpful to read this manual if you encounter any DOS errors.

### **WRITE PROTECTED**

You have a write-protect tab on the disk to which the Assembler is trying to write the output object file. This error typically appears near the beginning of pass 2 of your assembly.

#### **FILE NOT FOUND**

This is usually due to using the ASM command with a source filename that does not exist or is not on the disk in the current source drive and slot. It can also occur when a CHN, macro name, or Include command is used and the needed file is not present on the proper disk.

#### **I/O ERROR**

This is usually a read error but it can also be a write error caused by bad disk media. If it is a hard read error on an input file, the same problem will show up when you try to load that file. If it is a write error on the object file, that file appears in the catalog as being only one sector, or it results in an I/O error when you try to BLOAD the file into memory.

#### **DISK FULL**

This occurs when there is no space left on the output file disk for the output file. This can occur at any time during pass 2, so it is wise to determine if there is enough space on the disk for the output.

#### **NO BUFFERS AVAILABLE**

This error is not likely to occur unless you entered the system with MAXFILES set to 3. A normal assembly requires that there be at least two buffers available. If the Assembler has previously aborted an assembly and is unable to close all the open files, this situation could arise. Execute a direct DOS .CLOSE after this occurs to attempt to recover the file buffer space.

#### **FILE LOCKED**

This occurs if you locked the object file.

**By the Way:** If any other DOS error messages occur, probably the Editor/Assembler system is clobbered in memory, due to either a software bug or a hardware failure. Refer any repeatable errors of this kind to Apple Computer, Inc. in writing; if at all possible send a disk that will reproduce the problem. Include all relevant information: your machine type, memory size, peripheral cards installed, number of disk drives and controllers in use, and any modifications to this equipment.

Always attempt to recreate any problem with a fresh copy of the Assembler/Editor disk before concluding that you have found a program bug.

---

### **Assembler Syntax-Error Messages**

As the Assembler processes your assembly-language source file, it checks the syntax of each statement and your usage of identifiers and operands. It reads your source statements during each of its two assembly passes, and checks your syntax during both passes. Since different checks are performed during each pass, some error messages appear only during the first pass, some during the second, and some during both.

When the Assembler finds an error in a source statement, it prints an appropriate error message and skips to the next assembly statement. For this reason, the Assembler may not indicate all the syntax errors in your assembly statements the first time that you try to assemble your program. It may still find secondary errors in your source program after you correct all the errors flagged in a previous assembly.

You may sometimes see an Assembler syntax-error message that indicates one of a number of possible errors. To identify the exact problem, you may have to check the specific source statement that caused the error message.

As you watch your assembly, you may want to stop it if you see an error message that indicates a serious problem in your source file.

When the assembly is completed, the Assembler prints an error total at the end of the assembly listing. It is the sum of all the errors that were encountered during *both* assembly passes. This total may therefore be larger or even double the actual number of assembly statements that contain syntax errors.

#### **ADDRESS MODE ERROR**

The 6502 does not support all address modes for all opcodes. This error occurs when an invalid combination of opcode and address mode is found in a statement.

#### **ASSEMBLER PARAMETER ERROR**

The Assembler checks the parameters of the ASM command, passed from the Command Interpreter, for correct value ranges. This error occurs if the slot, drive or volume parameters are not within valid ranges. The error also occurs if a filename is more than 30 characters long, or if an invalid hex object address is encountered. This error message is associated with line 0 of an assembly.

#### **BRANCH RANGE ERROR**

The 6502 relative branch instruction has a limited addressing range. If the target address of a branch instruction is too far from the branch instruction, this error message is generated in pass 2.

#### **BUFFER SIZE ERROR**

The SBUFSIZ directive issues this message if the requested new buffer size would reduce the source buffer to a size smaller than the current source file residing in the buffer. The SBUFSIZ directive must be used to reduce buffer size only from a source file smaller than the desired reduced size.

#### **OVERFLOW ERROR**

The DS directive issues this error if the optional value expression is larger than eight bits. The expression directive characters can be used to ensure that this error is not generated.

#### **DIRECTIVE OPERAND ERROR**

This is a catch-all error message for many of the assembly directives that require specific operands. Generally this message indicates that an expression is not properly delimited by a required delimiter, such as a comma, but instead contains some other character where a delimiter is expected. For example, the statement below produces this error:

```
LABEL EQU 33 ; I FORGOT THE SPACE AFTER THE  
OPERAND
```

The data definition directives check to ensure that spaces do not follow commas when an expression list exists in an operand, issuing this error if the spaces are found.

The LST directive also issues this error if an invalid option character is encountered or if something other than a comma is used as a delimiter. The NO prefix for the LST options must be spelled exactly *NO* or *no*.

#### **DSECT/DEND ERROR**

This error indicates two things, depending upon the directive in the incorrect statement. This error is the result of attempting to nest the DSECT statement inside an active dummy section. The DSECT directive begins a dummy section, and the DEND directive terminates it. A second DSECT cannot be started while inside a dummy section. Terminating a dummy section with a DEND directive, when a dummy section is not active, also generates this error.



#### **DUPLICATE EXT/ENT ERROR**

The EXTRN and ENTRY directives check to ensure that a symbol being defined as an external or entry symbol is not already so defined in a prior EXTRN, ENTRY, or ZDEF statement. Any one symbol can only be defined once as having one of these special characteristics. In addition to this duplicate directive function, this error is issued if more than one external identifier exists in any operand expression.

#### **DUPLICATE IDENTIFIER ERROR**

This error occurs when an identifier in the label field of a statement is an exact duplicate of a previously defined identifier. Use another name for the identifier in the offending statement. You can receive DUPLICATE IDENTIFIER ERRORS for all occurrences of a duplicated identifier, although only some of these occurrences may require a correction. The symbol table dump does *not* show multiple entries for duplicate identifiers flagged with this error.

#### **EQUATE SYNTAX ERROR**

The Equate directive requires an identifier in the label field; this error occurs if the label field is empty—it is meaningless to define nothing as having some value. This error also occurs if the operand expression contains any external identifiers.

#### **EXPRESSION SYNTAX ERROR**

A valid term must follow all operators within an expression. A term in an expression is either a constant or an identifier. This error occurs when the text of a statement immediately following an operator, such as the addition operator (+), is not recognized as one of these two items. Terms are recognized by their first character:

First Character	Term Type
&	Used in macros
%	Binary constant
@	Octal constant
\$	Hexadecimal constant
'	ASCII character constant
.	Program counter reference
digit	Decimal constant
letter	Identifier (uppercase or lowercase)
#	Literal constant



This error also occurs if one of the radix characters used to begin numeric constants is not followed by any digits appropriate to that type of constant. For example, %4 generates this error since only 0 or 1 can follow the % (binary) radix character.

#### ***EXTRN USED AS ZXTRN WARNING***

This error indicates that an identifier, defined as a 16-bit external value by the EXTRN directive, was used as an 8-bit quantity, either as a zero-page address or as an immediate operand. This type of usage produces nonrelocatable object code if the external value becomes an absolute value when a linker attempts to edit the REL modules together.

The ZXTRN directive must be used in place of EXTRN for this type of identifier, and such identifiers must be defined using the ZDEF directive in the defining module (refer to ZDEF and ZREF for details).

#### ***INCLUDE/CHN NESTING ERROR***

The Include directive cannot occur as a statement within an included file or within a macro file. The CHN directive cannot occur within a macro file. This error results if you use either directive incorrectly.

#### ***INDEXING SYNTAX ERROR***

When specifying absolute-indexing or zero-page-indexing address modes, the addressing syntax required by the Assembler does not allow any character, except X or Y, to follow a comma. Refer to the section "Addressing Mode Summary" in Appendix B for details.

#### ***INDIRECT REQUIRES ZPAGE ERROR***

The Assembler checks the type of an expression in the indirect-indexed and indexed-indirect addressing modes. If the expression is not a zero-page expression, this error occurs. (The 6502 microprocessor requires the zero-page address.)

#### ***INDIRECT SYNTAX ERROR***

This error occurs if invalid syntax is used for indirect-indexed and indexed-indirect addressing modes. The most common mistake is the improper use of the X- and Y-registers, for example, LDA (expr,Y) and LDA (expr),X. This error also results from omission of the right parenthesis in the indirect addressing modes. Proper syntax is shown in the section "Addressing Mode Summary" in Appendix B.

#### **INVALID AFTER 1ST IDENTIFIER ERROR**

The SBUFSIZ and IBUGSIZ directives allow source and include buffer size management from the assembling program. These directives must be used before the first reference or definition of an identifier.

#### **INVALID DELIMITER ERROR**

This error message is the catch-all message for incorrect use of delimiters. All operand expressions must terminate with a space or a carriage return character. Various directives require the comma as a delimiter and do not allow spaces after commas. The indexing and indirect addressing syntaxes all require specific character delimiters. The delimiters are the space, the carriage return, the comma, and the parenthesis. When the assembler expects one of these characters and does not find it, this error results.

#### **INVALID FROM INCLUDE/MACRO ERROR**

The SBUFSIZ and IBUGSIZ directives must not occur in an include file or in a macro file.

#### **INVALID IDENTIFIER ERROR**

This error indicates one of two errors. First, an identifier contains a character not allowed within an identifier. Identifiers must begin with a letter and contain only letters, digits and/or the period. Second, an identifier is not allowed as the operand of a directive, usually because of a prior or conflicting definition. For example, an identifier cannot be used in the label field of an instruction and also be the operand of the EXTRN directive.

#### **INVALID WITH CORES ERROR**

The SBUFSIZ, IBUGSIZ, INCLUDE, MACLIB, and CHN directives cannot be used when you specify the coresident assembly option. Attempting to use these directives during coresident assembly aborts the assembly.

#### **MACRO ARGUMENT ERROR**

There are only 11 valid characters that can follow the macro argument character in the body of a macro. They are the ten digits and the uppercase letter X. This error results if any other characters are found following the & character during a macro expansion.

### **MACRO NESTING**

The Assembler does not support calling a macro from within a macro. Macros can only be called from either a source file or an included file.

### **OBJ BUFFER CONFLICT ERROR**

The OBJ directive requires that the placement of a coresident object code buffer be above the upper end of its symbol table and below the DOS 3.3 buffers. The DOS buffers end at \$9100 and the symbol table normally begins at \$3C00 unless modified by SBUFSIZ and IBUFSIZ.

### **OBJ BUFFER OVERFLOW ERROR**

The first OBJ directive in an assembly defines the lower limit of the OBJ buffer for an assembly. If the OBJ buffer fills up all the way to the DOS 3.3 file buffers, this error message results and the assembly is aborted.

### **OVERFLOW ERROR**

The ASCII source-to-binary conversion routines that convert numeric constants into binary values issue this error message if a constant definition generates more than 16-bits of information. For example, the constant \$33333 has too many digits; only four are allowed in a hexadecimal constant.

### **RELATIVE EXPRSN OPERATOR ERROR**

When the REL directive selects relocatable object-code output, the Assembler does not allow a relative address expression or subexpression to be divided, multiplied, ANDed, ORed, or exclusive-ORed; doing so causes the result to be nonrelocatable. This prohibition only applies if you used the REL directive in your source file.

### **RESERVED IDENTIFIER ERROR**

The Assembler does not allow the labels A, a, X, x, Y, and y as identifiers. If you insist on using these identifiers, there is a patch address offset in bytes 3 and 4 of the ASM file (using 0 base counting of code bytes) that indicates how far into the object code image to place a patch to cause this checking to be nullified. The patch consists of a CLC (\$18) and a RTS (\$60) opcode. They must be patched over the first two bytes of a JSR instruction.

#### **SLOT/DRIVE/VOLUME ERROR**

This error occurs if any of the named parameters are out of range on the file directives that use these parameters. Slot must be 1 through 7, drive 1 or 2, and Volume 0 through 254. This error is most commonly caused by entering 88 instead of 6.

#### **SYMBOL/RLD TABLE FULL ERROR**

This error occurs when insufficient space is available for the symbol table and, if the REL directive was used, for the relocation dictionary tables. The SBUFSIZ and IBUFSIZ directives default to a total size of 4K, but you can reduce these buffers to as little as one page each to provide additional symbol/RLD table space.

#### **SWEET16 OPCODE WARNING**

This warning indicates that a SWEET16 opcode was encountered in an assembly without at least one occurrence of the SW16 directive. This is a protective measure to prevent accidental use of SW16 mnemonics, which are very similar to 6502 mnemonics.

#### **SW16 REGISTER ERROR**

The SWEET16 pseudomachine has sixteen registers, designated R0 thru R15, and the Assembler validates that SWEET16 register expressions are only 4-bit results.

#### **UNDEFINED IDENTIFIER ERROR**

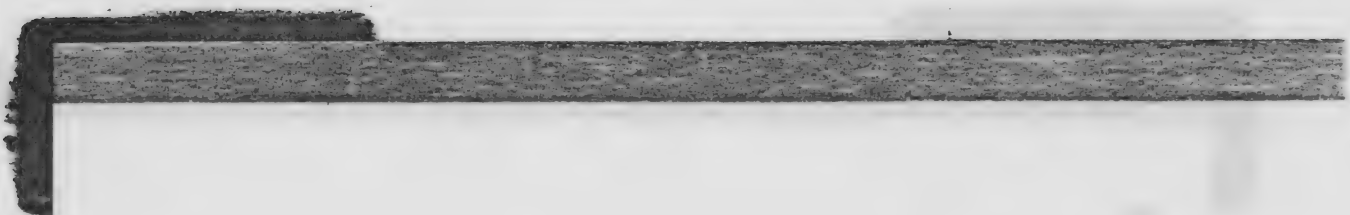
This error, generated in assembly pass 2, indicates an identifier was referenced, but never defined in your assembly source. It is most commonly caused by misspelling an identifier.

#### **UNDEFINED OPCODE ERROR**

This error occurs when macros are not enabled (the default condition), and a mnemonic is used that is not found in the Assembler's mnemonic table. It is most commonly a spelling error. This error becomes a DOS 3.3 FILE NOT FOUND error if you have enabled macros by using the MACLIB directive.

#### **>255 EXTRNS/ENTRYS ERROR**

When creating the Relocation Dictionary, the Assembler assigns a unique number to each EXTRN and ENTRY identifier; the numbers range from 1 to 255. The Assembler aborts if you use more than 255 such symbols in your source file.



[Faint, illegible text block consisting of several paragraphs of text, possibly a list or a series of entries.]

## Object File and Symbol Table Formats

### Object File Format

The Assembler generates two kinds of DOS object files: binary memory-image files and relocatable binary code files. Unless you use the REL assembly directive at the start of your assembly-language source file, the Assembler produces a binary file using the standard DOS binary format.

You can use the BLOAD command to load binary files produced by the Assembler, or, if your program is properly coded, you can BRUN your program from the normal BASIC/DOS environment (but *not* from within the Editor/Assembler system). To write your program so you can execute it using the BRUN command, your program must begin with executable code, not data, at the lowest address for which object code is generated.

The relocatable binary file is a file type not recognized by DOS. Table E-1 defines the format of this file type. In this table, the symbol = > means *indicates or implies*.

Table E-1. Relocatable File Format

Sector	Byte (Hex)	Contents of Byte
1	0	Starting RAM address, low byte
	1	Starting RAM address, high byte
	2	Length of RAM image, low byte
	3	Length of RAM image, high byte
	4	Length of code image, low byte
1 to M	5	Length of code image, high byte
	6 to c1+5	Binary code image, of length in bytes 4 and 5 above
	c1+6	Begin Relocation Dictionary, which consists of N 4-byte entries. N is variable (0 or greater). Each four bytes repeat the following structure:

Byte	Contents of Byte
1	<p>RLD-flags byte containing four flag bits as follows:</p> <p>\$80-bit Size of relocatable field Set =&gt; 2 bytes, Clear =&gt; 1 byte</p> <p>\$40-bit Upper/Lower 8 of a 16-bit value Set =&gt; high 8, Clear =&gt; low 8</p> <p>\$20-bit Normal/reversed 2-byte field Set =&gt; high-low, Clear =&gt; low-high (The DDB directive causes Set.)</p> <p>\$10-bit Field is EXTRN 16-bit reference Set =&gt; EXTRN, Clear =&gt; not EXTRN</p> <p>\$01-bit NOT END OF RLD flag bit Always Set on for RLD entry Clear marks end of RLD</p>
2	Field offset in code, low byte
3	Field offset in code, high byte
4	Low 8 bits of 16-bit value for an 8-bit field containing upper 8 bits. Zero if \$40-bit clear in RLD byte one. Or if the \$10-bit is set, then this is the ESD symbol number.
$N*4 + 1$	Binary 00 marks end of RLD.
$N*4 - 2$	Beginning of optional External Symbol Directory (ESD). This area contains bytes only if an EXTRN and/or ENTRY directive occurs in the program.
1 to sl	The EXTRN/ENTRY symbolic name of length sl bytes. All bytes have their \$80-bit set except the last one.
sl - 1	Symbol type flag byte defines which type of symbol EXTRN/ENTRY
\$10-bit	Set => EXTRN symbol type
\$08-bit	Set => ENTRY symbol type
sl + 2	EXTRN/ENTRY symbol number referred to by an RLD entry with EXT-bit set on.
sl + 3	High byte of offset for ENTRY type symbol. (low is in sl + 2)
End mark	Binary zero-byte marks end of the ESD entries, of which there may be zero.



## Symbol Table Formats

The symbol table generated during pass 1 of the assembly process is described here, along with the table format as it remains after the symbol table was modified by the symbol-table sort and print routine (pass 3). The symbol table is in its modified form and the RLD may be clobbered if the symbol table sort and dump is allowed to execute and it overwrites the RLD with its sort index table. The symbol table is a variable-length entry-format table with flag bits to signal the end of the variable-length-name character string.

The basic format is (Symbolicname)(Flagbyte)(Low value)  
(High value)

Symbolicname	consists of 1 to n characters, each with its \$80-bit set, except the last character's \$80-bit is reset.
Flagbyte	contains the bits that define the characteristics of the symbol, its value, and how it can be used to generate instructions, as follows:
• \$80-bit	Undefined-Symbol bit  This means that the symbol is referenced but not defined. This flag is reset when a symbol is defined; if it remains set at the end of pass 1, the symbol is undefined and causes the NO SUCH LABEL error during pass 2. Symbols with this bit set are printed by pass 3 with an * next to the address (which is meaningless; it is simply the value of the program counter at the first reference).
\$40-bit	Unreferenced-Symbol bit  The symbol is defined but never used as the operand of any instruction in the program. This bit causes the ? to print next to the address value for an unreferenced symbol in the dump.
\$20-bit	Relative-Symbol bit  The symbol's value is a relative symbol rather than an absolute address. Relative means relative to the beginning of the module. It is used internally by the assembler, when generating the Relocatable type of output file, to cause an RLD entry to be created for any references to the symbol.
\$10-bit	External-Symbol bit  The symbol is defined as an external symbol via the EXTRN directive. This causes the symbol to be put into the ESD and prevents the symbol from being considered undefined, even though no value is assigned to it. Using such a symbol causes an RLD entry to be marked as EXT and causes the external symbol number to be put in the RLD entry in place of the relative offset. External symbols can only represent undefined 16-bit values (not 8-bit or zero-page values).

---

\$08-bit	ENTRY-Symbol bit
	The symbol is an entry point into the module that can be referred to by an EXTRN in another module. This causes the symbol to be included in the ESD for resolution by a linkage editor. Note: A linkage editor is not implemented.
\$04-bit	Macro-name bit
	This bit is reserved for signalling whether the symbol is actually a macro filename: the value bytes hold drive and slot, respectively. Note: this feature is not implemented.
\$02-bit	NO-Such-Label Error bit
	The symbol has caused one or more NO SUCH LABEL errors. This is used to prevent a duplication of a single error in the error summary table during pass 2.
\$01-bit	Forward-Referenced bit
	A forward reference forced the symbol to be considered a 16-bit value. Zero-page labels print in the symbol dump with blanks for the first two bytes. They print with two zeros when this bit is set. If the definition is moved forward so the symbol is defined before it is referred to, reassembling the program generates shorter zero-page instructions.

---

When the Symbol Sort and Dump routine executes, it modifies the symbol table format to speed up the scanning of the table for its second phase. The last character of each symbol has its high-order bit set on and the Flagbyte is changed. If the Flagbyte has its \$80-bit set, its value is changed by ORing it with \$7E to set all but the \$01-bit on. The \$01-bit is retained, and the \$80-bit is set off to mark the end of the symbolic name. Thus, if pass 3 is run, all Flagbytes have their \$80-bits reset, and undefined symbols have a Flagbyte of \$7E or \$7F.

## Editing BASIC Programs

Using the Editor, you can perform many useful editing functions on the text of your BASIC programs. You can use the Find command to locate all statements referring to a given BASIC variable or line number, for example. You can use the global Change command to change all occurrences of a variable name to a new name, or to change all occurrences of a GOSUB to some other line number.

### Using the Editor to Edit BASIC Programs

To use the Editor to edit BASIC programs, you must first convert your BASIC program into a sequential text file. The *DOS Programmer's Manual* contains a section "Capturing Lines of a BASIC Program" explaining how you can do this. After you convert your BASIC program into a sequential text file, you can then load the text using the Editor and edit the program as you would any other text file.

**Remember:** Before you edit your program, set the Tabs command to zero to keep the left margin of your text at the left side of the screen.

When you are editing BASIC programs, you must be careful not to change the line number of a statement without also changing all the references to it elsewhere in your program. If you want to change any line numbers within a BASIC program, it is better to use the Applesoft/Integer BASIC RENUMBER programs instead of using the Editor.

When you are editing a BASIC program using the Editor, the Editor shows two line numbers on every line. The first line number is the Editor's relative line number; and the second is the line number of the BASIC statement. Only the line number of the BASIC statement is actually stored in the text file. As you are editing, you can use only the Editor's relative line numbers as the line number parameters for Editor commands.

When you finish editing, you must save the text file back on your disk and reenter BASIC, using the Editor End command. To reenter your edited text file into BASIC, type the command

```
EXEC myprogram
```

where *myprogram* is the name of your BASIC program text file. This command causes DOS to read the entire text file from the disk into BASIC, just as if you typed it from the keyboard.

Since BASIC places each line from your text file into a BASIC program according to its line number, each line of your BASIC text must begin with a line number. The order of the lines within your text file is not important; changing the order of the lines without changing their line numbers will not change the way your program is interpreted by BASIC.

## The SWEET16 Interpreter

### A 16-Bit Pseudomachine

The Assembler can generate object code for the 16-bit pseudomachine known to experienced Apple II users as SWEET16. This pseudomachine is described in the October 1977 issue of BYTE magazine in the article entitled *SWEET16: The 6502 Dream Machine*. The Apple II Integer BASIC ROM contains a SWEET16 interpreter that implements this 16-bit pseudomachine in software.

The Editor contains a SWEET16 interpreter.

You can take advantage of this 16-bit processor in your own assembly-language programs by capturing SWEET16 interpreter code from the Editor in your own code; this allows you to use the SWEET16 routines on any type of Apple II. To use SWEET16, your program must execute a subroutine call to the starting address of the SWEET16 software interpreter. This starting address is at location \$F689 in the Apple II Integer Basic ROM. When you are using the Assembler, you can generate this subroutine call automatically by using the SW16 Assembly directive described in Chapter 3.

Immediately following this call to the SWEET16 interpreter, your program code should contain only SWEET16 instructions, to be *executed* in sequence by the SWEET16 interpreter. To return control of your program to the 6502 microprocessor, you must include the SWEET16 RTN operation code at the end of each sequence of SWEET16 instructions. Following this RTN operation code, your program must contain only 6502 instructions, since the 6502 microprocessor resumes executing your program directly.

## SWEET16 Interpreter Operation

When your program calls the SWEET16 interpreter, the interpreter saves the current 6502 register contents and begins interpreting the SWEET16 instructions starting at the address immediately following your SWEET16 subroutine call. These instructions can manipulate data in memory, manipulate SWEET16's own internal registers, or jump to other program locations.

At the end of the sequence of SWEET16 instructions, your program must contain a SWEET16 RTN instruction. When the SWEET16 interpreter finds this instruction, the interpreter restores the original 6502 register contents and returns the 6502 microprocessor to the instruction immediately following the SWEET16 RTN. The 6502 microprocessor then resumes normal execution of 6502 instructions.

A simple use of the SWEET16 interpreter is demonstrated in this example:

0900:09 00 02	10	LDA IN.Y	:GET A CHARACTER
0903:09 00	11	CMP #'M'	:M IS FOR MOWE?
0905:00 09	12	BNE NOTMWE	:NO, DON'T MWE
0907:20 09 F5	13	SW16	:SWEET ON ME?
090A:41	14	MOVLP LDI R1	:GET SOURCE BYTE
090B:52	15	STI R2	:PUT TO TARGET
090C:F3	16	DCR R3	:COUNT DOWN LENGTH
090D:07 F5	17	BNE MOVLP	: UNTIL ZERO LENGTH
090F:00	18	RTN	:BACK TO 6502
0910:03 05	19	NOTMWE CMP #'E'	:ALL DONE?
0912:00 13	20	BEQ EXIT	:YES, GET OUT
0914:03	21	INY	:NO, CONTINUE

The SWEET16 instructions appear on lines 14 through 18, while the remainder of the instructions are normal 6502 instructions. Note that the SW16 directive simply generates a 6502 JSR instruction to the address \$F689.

The SWEET16 instruction mnemonics and operation codes are different from those of standard 6502 instructions. You must be careful not to intermix SWEET16 instructions with 6502 instructions in your programs except as outlined above: every block of SWEET16 instructions must be preceeded by a call to the SWEET16 interpreter and must be ended with a SWEET16 RTN instruction. The Assembler helps you avoid using SWEET16 instructions unintentionally by issuing the SWEET16 OP0000E warning if you should use a SWEET16 instruction in your program without somewhere also including the SW16 Assembly directive.

## **SWEET16 Instruction Summary**

### **Register Instructions**

OpCode	Mnemonic	Explanation
1n	SET Rn,Const	Load Rn with the two byte constant
2n	LD Rn	Load R0 from Rn
3n	ST Rn	Store R0 to Rn
4n	LDI Rn	Load low byte of R0 from address in Rn
5n	STI Rn	Store low byte of R0 to address in Rn
6n	LDD Rn	Load R0 with word at address in Rn
7n	STD Rn	Store R0 to word at address in Rn
8n	LDP Rn	Decrement Rn, load low byte of R0 indirect Rn
9n	STP Rn	Decrement Rn, store low byte of R0 indirect Rn
An	ADD Rn	Add Rn to R0, storing result in R0; set status
Bn	SUB Rn	Subtract Rn from R0, result in R0; set status
Cn	POPD Rn	Decrement Rn and load R0 from address in Rn
Dn	CPR Rn	Compare R0 to contents of Rn; set status
En	INR Rn	Increment the contents of Rn by 1
Fn	DCR Rn	Decrement the contents of Rn by 1

### **Control Instructions**

OpCode	Mnemonic	Explanation
00	RTN	Return to 6502 mode
01	BR ea	Branch to the effective address (ea)
02	BNC ea	Branch if No Carry
03	BC ea	Branch if Carry
04	BP ea	Branch if Plus
05	BM ea	Branch if Minus
06	BZ ea	Branch if Zero



---

07	BNZ ea	Branch if NonZero
08	BM1 ea	Branch if Minus 1
09	BNM1 ea	Branch if Not Minus 1
0A	BK	Break into 6502 Monitor
0B	RS	Return from SWEET16 subroutine
0C	BS ea	Branch to Subroutine
0D	CPIM const	Compare R0 to two-byte constant; set status
0E	BRL ea	Branch long to effective address
0F	BSL ea	Branch Subroutine Long to effective address

---

The mnemonic definitions for the SWEET16 LDI and STI instructions differ from those originally defined in the BYTE article, because the original LD @ and ST @ mnemonics conflict with the Assembler's use of the @ character to indicate octal constants. In addition, the STP mnemonic is used in place of POP, to be consistent with the LDP mnemonic (the Assembler also accepts POP).

The last three instructions are new instructions that are not recognized by the Integer BASIC version of the SWEET16 interpreter. These instructions are recognized by the SWEET16 interpreter that is contained in the Editor module. This interpreter resides in the Language Card from address \$DC2E through \$DE93, when the Editor is in memory. You therefore have two versions of the SWEET16 interpreter to choose from, if you don't mind including the Editor SWEET16 interpreter into your program code:

- The Integer BASIC version in ROM, designed to conserve ROM memory, at the expense of execution speed
- The Editor version, designed for execution speed without regard to code size.

## System Memory Usage

### The Editor/Assembler

Figure H-1 shows the memory areas used by the various modules that comprise the Editor and Assembler. In the 64K version of the Editor/Assembler, both the Editor and the Assembler are in memory at the same time. In the 48K version of the Editor/Assembler, the Assembler shares the same memory space with the Editor. When you call the Assembler in a 48K system, the Assembler code overlays the Editor's text buffer, as shown in Figure H-1.

This memory map of the 64K system is provided for reference only. Apple Computer, Inc. reserves the right to change or expand the areas used at any time and without notice.

Figure H-1. Memory Map of the 64K Editor/Assembler

10000	Editor SWEET16 registers
1000A	Editor LOWEM, HIGHM, and
1000F	Text End pointers (5 bytes)
100FF	Editor/Assembler zero page
10100	Editor/Assembler
101FF	6502 stack area
10200	Editor input buffer area
102FF	
10300	Editor and assembler parameters
103FF	and DOS 3.3 Jump vectors, etc.
10400	Editor and Assembler text
107FF	screen display page 1
10800	Editor parameter save areas
108FF	and Assembler text screen page 3
10C00	EDASM command interpreter
114FF	module and editor work area
11500	Assembler code and data
127FF	tables, messages and buffers
13000	Assembler buffers and tables (to 26FF)
13100	and Editor TEXT BUFFER (to 3100) (Note Overlap)
19145	DOS 3.3 with 5 file buffers
18FFF	end of DOS 3.3 and main memory
Language Card	
1D000	Editor Code
1D000	Assembler Code
1DFFF	Card 4K fold (\$C080)
1DFFF	in 4K fold (\$C080)
1E000	Remainder of Assembler object code in Language Card
1F7FF	
1F800	Auto-Start Monitor ROM image
1FFFF	Reset Vectors

## The Bugbyter Debugger

Figure H-2 shows the memory areas in which you can load the Bugbyter debugging program. Bugbyter occupies 1.6K of contiguous memory and must be located so it does not overlap any of your own program's code or data areas. Chapter 4 includes a description of how you can relocate the Bugbyter to any of the areas shown in the figure.

**Figure H-2.** Memory Map Showing Locations of Bugbyter and Your Program

48K motherboard RAM:		Optional Language or RAM Card:	
\$BFFF: +	+ \$FFFF: +	+	
	DOS	Monitor	
\$9600: +	+ \$F800: -	+	
	Bugbyter can reside anywhere in here	Bugbyter can reside anywhere in here	+ undefined +
	\$D000: +	bank 2	+ + bank 1 +
\$800: +	+		
	text screen		
\$400: +	-		
\$200: +	+		
	stack		
\$100: +	+ < first \$20 bytes reserved (\$100-11F)		
	zero page		
0: +	+		

Note: Bugbyter reserves the last 32 bytes of the program stack.

## **Editor/Assembler File Components**

### **The EDASM Files**

When you run the Editor/Assembler, the EDASM program determines whether you are currently using a 48K or a 64K Apple II, and loads the appropriate code portions for your particular machine. (You may have noticed that there are two sets of object modules for the EDASM program on your DOS Programmer's Tool Kit Volume II disk.) If you ever want to move the Editor/Assembler to another disk, you can save space by copying only the EDASM modules that you need for your particular Apple II configuration:

- For a 48K Apple II, the Editor/Assembler modules you need are:

- EDASM
- EDASM.1
- EDASM48.2
- EDASM48.3
- EDASM48.4
- EDASM48.5
- EDASM48.6

- For a 64K Apple II, the Editor/Assembler modules you need are:

- EDASM
- EDASM.1
- EDASM64.2
- EDASM64.3
- EDASM64.4
- EDASM64.5
- EDASM64.6

See the INIT command in the *DOS User's Manual*. See also the section on making a turnkey disk in the *DOS Programmer's Manual*.

### **Making a HELLO Program**

If you want, you can configure your programming system disk so that whenever you boot your system disk, you automatically start up the Editor/Assembler program. This is done by making the EDASM program into your system disk's greeting program or turnkey program.

If you recall, everytime you initialize a disk, you must specify the name of a greeting program. Typically, this greeting program is named HELLO. Whenever you boot a disk that has a greeting program, your Apple II automatically runs this BASIC program as the first thing after loading DOS itself.

To set up EDASM as your disk's greeting program, all you need do is rename EDASM with the name of your disk's greeting program. If you are initializing a new disk, you can simply specify the name of the greeting program to be EDASM. Once the Editor/Assembler program has the same file name as your disk's greeting program, you are automatically placed at the Editor command level whenever you boot this disk.



# Index

## A

- A-Register 110, 146
- absolute identifier(s) 67
- absolute location in memory, identifying 45
- absolute ORG directive 75
- activating a printer 34
- Add command 15, 29, 176
- ADDRESS MODE error 199
- address-mode syntax 51
- All ASSEMBLY ID 168
- Alphabetic Symbol Table
  - option 94
- ALS Smarterm 80-Column display
  - card xiv
- &O parameter 99
- &X parameter 100
- Append command 26, 175
- Apple II SWEET16 routine(s) 78
- Apple II system, 48K 12
- Apple IIe, 80-Column Text
  - Card xiv
- arithmetic operator(s) 70
- ASC directive 84
- ASCII directive See ASC directive
- ASM srcfile command 178
- ASMIDSTAMP 12
- ASMIDSTAMP file 59
- Assembler
  - ASC directive 84
  - control character(s) 29
  - Date directive 85
  - DCI directive 84
  - DDB directive 82
  - DEF directive 79-80
  - defining data 81
  - DEND directive 76
  - DFB directive 82
  - DOS error messages 197-198
  - DS directive 83
  - DSECT directive 75-76
  - dummy section(s) 75-76
  - DW directive 82
  - entry point symbol(s) 64
  - EQU directive 79
  - error messages 197-205
  - external symbol(s) 64
  - IDNUM directive 85
  - listing cycle times 93
  - location counter 71-72
  - modular assembly 88
  - MSB directive 83
  - no object code 56, 76
  - OBJ directive 77
  - ORG directive 74-75
  - program counter 71-72
  - REF directive 80
  - REL directive 77-78, 160
  - STR directive 84
  - suppressing generation of object file(s) 56
  - SW16 directive 78
  - syntax-error messages 199-205
  - tab character(s) 43, 66
  - zero-page address 65
  - ZXTRN directive 81
- Assembler directives 65, 67
- Assembler ID stamp 12, 59
- ASSEMBLER PARAMETER ERROR 199
- assembling programs from memory 57
- assembling your program 53



- Assembly directives 73-96
  - DEND directive 76
  - DSECT directive 75-76
  - OBJ directive 77
  - ORG directive 74-75
    - absolute 75
    - relative 74
  - REL directive 77-78
  - SW16 directive 78
- assembly-language
  - program(s) 3
  - programming 3
  - source file(s) 3, 5, 65-72
  - instruction(s) 5, 67
  - mnemonics 51
  - subroutine(s) 6
- assembly listing(s) 51, 54
  - Alphabetic Symbol Table
    - option 94
  - Assembler warnings 93
    - generating 59
  - header(s) 62, 96
  - interpreting 61-62
  - interrupting 63
  - macro expansion(s) 94, 99
  - making your listing readable 95
  - single-stepping 63
  - tab settings 63
  - title(s) 96
  - turning off 63-64
  - Value-Ordered Symbol Table
    - option 94
- Assembly macros 67, 68, 97
  - &O parameter 99
  - &X parameter 100
- Assembly statement(s)
  - comment field 66
  - label field 66-67
  - mnemonic field 66-67
  - operand field 66
  - operation field 66
  - syntax 65-66
- B**
  - backing up your work 28
  - backup files 28
  - BASIC subroutine(s) 6
  - binary file(s) 5
  - binary object program(s) 52
  - binary-logic operator(s) 70
  - Breakpoint subdisplay 143
  - BRANCH RANGE ERROR 200
  - Breakpoint Flag(s) 111
  - Breakpoint subdisplay 113
    - breakpoint(s) 142
      - real 142, 148
      - transparent 142
    - BUFFER ERROR 196
    - BUFFER SIZE ERROR 200
    - bug(s) 105
    - Bugbyter
      - altering Master Display
        - layout 137-138
      - Breakpoint Flag(s) 111, 148
      - Breakpoint subdisplay 113
      - changing registers 136
      - Code Disassembly
        - subdisplay 112, 118
      - command level 114-115
      - command line 114
      - counting instruction cycles 147
      - Cycle Count Register 111, 147
      - display option(s) 145-146
      - editing commands 116
      - entering DOS commands 117
      - entering the Monitor 131
      - executing undefined
        - opcodes 156
      - keyboard interrupt
        - character 153
      - Master Display 109
      - Memory Cell subdisplay 113
      - Memory Page display 128
      - memory restrictions 107-108
      - Memory subdisplay 122
      - Off command 153
      - Quit command 128
      - Register Subdisplay 110
      - relocating 130
      - restarting 131
      - Single-Step operation(s) 140
      - single-stepping your
        - program 119
      - soft switch(es) 151, 154
      - Stack subdisplay 111
      - Trace mode 125
      - Trace operation(s) 140
      - Trace Rate Register 111
      - turning off display 153
      - turning off keyboard
        - polling 154
      - using in a Language Card 130
      - using the keyboard 153
      - viewing memory 127, 132
    - Bugbyter Execution mode 115
    - Bugbyter Registers 111
    - Bugbyter Trace mode 115
    - button 0 See hand control
      - button 0

## C

- calling the Assembler 55-56
- CAPS LOCK command 23
- caps lock mode 23
- CAT command 18, 175
- Catalog command See CAT command
- Chain directive See CHN directive
- Change command 36, 177
- changing registers 136
- changing text 36
- changing the command delimiter 37
- changing the current disk 28
- changing the tab settings 42
- Character directive See CHR directive
- CHN directive 88
- CHR directive 95
- clearing the text buffer 19, 32
- CLR command 144
- CMD SYNTAX ERROR 196
- Code Disassembly
  - subdisplay 112, 118
- Column 80 command 177
- Column 40 command 177
- command delimiter 37
- command level 114-115
- command line 114
- command names 22
- command(s), more than one on a line 22
- command-error messages 195-197
- comment field 66, 72-73
- conditional assembly 85-88
- conditional assembly block 86
- conditional-assembly directive(s) 85-88
- constant(s) 69
  - numeric 69-70
  - string 69-70
- control character(s) 29
- CONTROL**-(**E**) command 23, 177
- CONTROL**-(**R**) command 33, 176
- CONTROL**-(**W**) command 23, 177
- Copy command 31, 176
- copying lines 31
- coresident assembly 57
- counting instruction cycles 147
- current disk slot 25
- current drive 25
- current filename 25
- current volume 25

- cursor 14
- Cycle Count Register 111, 147
- Cycle Times option 93

## D

- Date directive 85
- DB directive 82
- DCI directive 84
- DDB directive 82
- debugging 105
  - real-time code 150-152
- DEF directive 79-80
- default tab settings 16
- Define Byte directive
  - See DFB directive
- Define Double Byte directive
  - See DDB directive
- Define Storage directive
  - See DS directive
- Define Word directive
  - See DW directive
- defining data 81
- Delete command 30, 31, 176
- DEND directive 76
- device-control-string 61
- DFB directive 82
- direct DOS commands,
  - executing 24
- DIRECTIVE OPERAND error 200
- directive(s) 52
- DISK FULL error 194, 198
- display option(s) 145-146
- displaying text 16
- DO directive 85-87
- DOS commands 24, 117, 175, 180
- DOS error messages 197-198
- DOS EXEC file 61
- Drive command 28, 176
- DS directive 83
- DSECT directive 75-76
- DSECT/DEND error 200
- dummy section(s) 75-76
- DUPLICATE EXT/ENT error 201
- DUPLICATE IDENTIFIER error 201
- DW directive 82

## E

- Edit command 38, 177
- Edit mode 17, 38-40
- editing commands 116
- editing two files at once 41

## Editor

- activating a printer 34
- Add command 29
- backing up your work 28
- backup files 28
- CAPS LOCK command 23
- Change command 36
- changing text 36
- changing the current disk 28
- changing the tab settings 42
- command names 22
- CONTROL**-**E** command 23
- CONTROL**-**R** command 33
- CONTROL**-**W** command 23
- copying lines 31
- Delete command 31
- Edit command 38
- Edit mode 38-40
- entering DOS commands 117
- entering text 15
- entering the Monitor 45, 131
- error messages 193-197
- executing direct DOS commands 24
- executing undefined opcodes 156
- 48K Apple II system 12, 219
- help 23
- Insert command 29
- inserting files 26
- joining files 26
- listing cycle times 93
- listing lines of text 32
- Load command 26
- loading files from disk 26
- lowercase characters 23
- maximum file size 11
- moving lines 31
- printing lines of text 34
- re-entering 45
- relative line number(s) 16
- relisting lines of text 33
- restoring the text buffer 32, 45
- Save command 27
- saving files 18, 27
- searching text 35
- Set Lcase command 23
- Set Lowercase command 23
- Set Ucase command 23
- Set Uppercase command 23
- split-buffer mode 41
- starting the Editor 12
- text buffer 11, 15
- text size 25
- typing commands 22
- using a 40-column display 43
- viewing the disk contents 28
- Editor command level 13, 14, 19, 22
- Editor program 11
- effective address(es) 146
- 80-column text card xiv, 21
- ELSE directive 86-87
- End command 21, 44, 178
- entering DOS commands 117
- entering text 15
- entering the Monitor 45, 131
- ENTRY directive 80
- entry point symbol(s) 64
- EQU directive 79
- Equate directive See EQU directive
- EQUATE SYNTAX error 201
- ERR: BAD FORMAT 195
- ERR: BAD RANGE 195
- ERR: BAD SLOT/DRIVE/VOL # 195
- ERR: MEMORY FULL 196
- ERR: SYNTAX 196
- error messages 193-197
- executable object program(s) 3, 51
- executing direct DOS commands 24
- executing undefined opcodes 156
- Execution mode 148
- EXPRESSION SYNTAX ERROR 201
- expression(s) 70
- external directive See EXTRN directive
- external symbol(s) 64
- EXTRN directive 80
- EXTRN USED AS ZXTRN warning 202

## F

- FAIL directive 87-88
- field(s)
  - comment 66, 72-73
  - label 66-67
  - mnemonic 66, 67-68
  - operand 66, 68-72
  - operation 66, 67-68
- File command 15, 25, 175
- FILE LOCKED error 194, 198

FILE NOT FOUND error 194, 198  
FILE TYPE MISMATCH 195  
file(s)

binary 5  
storing 18  
text 18

FIN directive 85-87  
Find command 35, 177  
40-column display 43  
48K Apple II system 12, 219

## G

G command 149  
game paddle 0 See hand control 0  
generating assembly listings 59  
>255 EXTRNS/ENTRYS error 205

## H

hand control 0 155  
hand control button 0 154  
header(s), assembly listing 62  
help 23  
high-byte operator(s) 70-71  
HIMEM 163

## I

I/O error 194, 198  
IBUFSIZ directive 89  
identification routine(s) 168  
identifier(s) 5, 66, 68-69  
absolute 67  
zero page 67  
identifying absolute location in memory 45  
IDNUM directive 85  
IF directive 85-87  
IN command 148  
INCLUDE directive 89  
INCLUDE/CHN NESTING error 202  
indexing syntax error 202  
input mode 15, 29  
insert command 29, 176  
inserting files 26  
instruction statement(s) 65  
instructions,  
assembly-language 5  
interpreting assembly listings 61-62  
interrupt character 153  
interrupting assembly listings 63

INVALID DELIMITER error 203  
INVALID IDENTIFIER error 203  
INVALID WITH CORES error 203

## J

J command 150  
joining files 26

## K

keyboard interrupt character 153  
keyboard polling, turning off 154  
Keyboard-Polling soft switch 154  
KILL2 command 42, 177

## L

L command 118  
label field 66-67  
Language Card 130  
Length command 25, 175  
line number(s), relative 16, 22  
List command 32, 176  
Listing directive See LST directive  
listing lines of text 32  
Load command 19, 26  
Load frame command 175  
loading files from disk 26  
location counter 71-72  
logical-page length 60  
low-byte operator(s) 70-71  
lowercase character(s) xiv, 23  
LST directive 92

## M

machine instruction code(s) 5  
Machine-instruction byte(s) 146  
MACLIB directive 90, 96  
MACRO ARGUMENT ERROR 203  
macro definition file(s) 97  
MACRO NESTING error 204  
macro parameter(s) 97  
Master Display 109  
Master Display layout, altering 137-138  
MEM command 122, 133  
memory  
absolute location 45  
altering 135  
viewing 132  
Memory Cell subdisplay 113, 132-133  
Memory Page display 128, 132, 133

Memory Page Display mode 115  
 Memory subdisplay 122, 132  
 mnemonic field 66, 67-68  
 mnemonic(s) 5, 67  
   assembly-language 51  
 modular assembly 88  
 Mon command 178  
 Monitor, entering 45, 131  
 more than one command on a line 22  
 Most Significant Bit directive  
   See MSB directive  
 moving lines 31  
 MSB directive 83  
 MULTI BUFFER ERROR 196

## N

New command 19, 32, 176  
 NO BUFFERS AVAILABLE  
   error 195, 198  
 no object code 56, 76  
 number of parameters 99  
 numeric constant(s) 69  
 numeric expression(s) 70  
 NUMERIC OVERFLOW error 196

## O

OBJ BUFFER CONFLICT  
   error 204  
 OBJ BUFFER OVERFLOW  
   error 204  
 OBJ directive 77  
   object code, relocatable 77-78  
   object file(s), suppressing  
     generation of 56  
   object program(s)  
     binary 52  
     executable 51  
     relocatable 52  
   Off command 153  
   On command 153  
   opcode 51  
   operand 5  
   operand field 66, 68-72  
   operation field 66, 67-68  
   operator(s)  
     arithmetic 70  
     binary-logic 70  
     high-byte 70-71  
     low byte 70-71  
   optional character(s) 22  
 ORG directive 74-75  
   absolute 75  
   relative 74

origin address 74  
 OUT command 149  
 OVERFLOW error 200, 204

## P

paddle  
   See hand control  
     button 0  
 Page directive 91  
 PARAMETER(S) OMITTED  
   error 196  
 physical-page length 61  
 PR# slot# command 178  
 Print command 16, 32, 34, 176  
 printing lines of text 34  
 Processor Status Byte 146  
 Processor Status Register 110  
 program(s) 3  
 program bug(s) 105  
 Program Counter 110  
 program starting address 74  
 PROGRAM TOO LARGE  
   error 195  
 programming guides xiii  
 programming process 4  
 prompt character 22  
 pseudo-operation(s) 68

## Q

Quit command 128

## R

R command 142  
 RBOOT routine 160, 161  
 re-entering the Editor 45  
 real breakpoint(s) 142, 148  
 REF directive 80  
 real-time code, debugging  
   150-152  
 reference card 22  
 reference manuals xiii  
 Register Subdisplay 110  
 register(s) 68  
   altering 136  
   Bugbyter 111  
 REL directive 77-78, 160  
 RELATIVE EXPRSN OPERATOR  
   error 204  
 relative line number(s) 16, 22, 31  
 relative ORG directive 74  
 relisting lines of text 33  
 relocatable module(s) 160  
 relocatable object code 77-78  
 relocatable object program(s) 52  
 relocating Bugbyter 130



- Relocating Loader 52
  - restrictions 160-161
  - saving HIMEM 163
- Relocating Loader program 77-78
- REP directive 95
- Repeat directive See REP directive
- Replace command 31, 176
- RESERVED IDENTIFIER
  - error 204
  - starting Bugbyter 131
  - storing the text buffer 32, 45
  - LOAD program 77
  - LOAD routine 160, 162

## S

- S command 119
- Save command 18, 27, 175
- saving files 18, 27
- SBTL directive 62, 96
- SBUFSIZ directive 89
- search and replace 36
- searching text 35
- selecting a printer 59
- SET command 138-139
- Set Delim command 177
- Set Lcase command 23, 178
- Set Lowercase command
  - See Set Lcase command
- Set Ucase command 23, 178
- Set Uppercase command
  - See Set Ucase command
- Single-Step mode 115, 139
- Single-Step operation(s) 140
- single-stepping
  - through assembly listings 63
  - your program 119
- Skip directive See SKP directive
- SKP directive 96
- Slot command 28, 176
- SLOT/DRIVE/VOLUME
  - ERROR 205
- soft switch(es) 151, 156
- source file directives 88-91
- source file(s), assembly-
  - language 3, 5
- split-buffer mode 41
- Stack Pointer 110, 146
- Stack subdisplay 111
- starting the Editor 12
- storing files 18
- STR directive 84
- string constant(s) 69-70
- String directive
  - See STR directive

- subroutine(s)
  - assembly-language 6
  - BASIC 6
- Subtitle directive
  - See SBTL directive
- suppressing generation of object
  - file(s) 56
- SW16 directive 78, 214
- SW16 REGISTER ERROR 205
- Swap command 41, 177
- SWEET16 interpreter 213
- SWEET16 OP CODE warning 205
- symbol table listing(s) 51, 64-65
- SYMBOL/RD TABLE FULL
  - error 205
- symbolic name 5
- symbols 5
- syntax
  - address-mode 51
  - of assembly statement(s) 65-66
- SYNTAX ERROR 194
- syntax-error messages 199-205

## T

- tab
  - character(s) 15, 16, 42, 66
  - settings 16, 63
  - changing 42
- Tab command 43, 178
- text
  - displaying 16
  - entering 15
- text buffer 15, 176
  - clearing 19, 32
  - Editor 11
  - restoring 32
- text file(s) 18
- Trace mode 125, 139, 141
- Trace operation(s) 140
- Trace Rate Register 111, 145
- tracing subroutines 142
- transparent breakpoint(s) 142
- Truncate-off command
  - See TruncOff command
- Truncate-on command
  - See TruncOn command
- TruncOff command 44, 178
- TruncOn command 44, 178
- turning off assembly listing(s) 63-64
- turning off display 153
- turning off keyboard polling 154
- tutorials 6

## U

Unassembled Source option 94  
UNDEFINED IDENTIFIER  
error 205  
UNDEFINED OP CODE error 205  
undefined opcode(s) 156  
UNKNOWN COMMAND  
error 197  
using a 40-column display 43  
Using macros 96-100

## V

Value-Ordered Symbol Table  
option 94  
viewing disk contents 28  
Volume 0 25  
Volume command 29, 176

## W

Where line command 178  
wildcard character(s) 35  
wildcard(s) 25  
WRITE PROTECTED error 194,  
197

## X

X-Register 110, 146

## Y

Y-Register 110, 146

## Z

ZDEF directive 80  
zero page identifier(s) 67  
zero-page address 65  
zero-page addressing 72  
ZREF directive 81  
ZXTRN directive 81